Miku Laitinen

Jatkuvan integraation käyttöönotto

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tietotekniikka
Insinöörityö
16.4.2011

Metropolia

Tiivistelmä

Tekijä(t) Otsikko Sivumäärä Aika	Miku Laitinen Jatkuvan integraation käyttöönotto 39 sivua + 1 liite 16.4.2011
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja(t)	teknologiajohtaja Jari Eriksson lehtori Outi Grotenfelt

Tämän insinöörityön tavoitteena oli suunnitella ja ottaa käyttöön jatkuva integraatioprosessi erästä Playground Finland Oy:n web-sovellusta varten. Sovellus on kehitetty tiedonkeruuta, hallintaa ja jakamista varten, ja se on tällä hetkellä muutamalla asiakkaalla testikäytössä. Sovellus on toteutettu Javalla suomalaista Vaadin-sovelluskehystä käyttäen.

Työn lähtökohtana olleen sovelluksen ja sen kehitysprosessin kompleksisuus alkoi kasvaa asiakasmäärän ja sovelluksen asiakaskohtaisten toiminnallisuuksien lisääntyessä. Lisääntyneen manuaalisen työn ja siitä johtuneen kasvaneen virhealttiuden vähentämiseksi päätettiin sovelluksen kehitys- ja kokoamisprosessia automatisoida ottamalla jatkuva integraatio käyttöön.

Insinöörityön ensimmäinen vaihe oli prosessiin ja sen käyttöönottoon tarvittavien työkalujen soveltuvuuden arviointi. Työhön parhaiten soveltuneet työkalut esiteltiin ja niiden valinnat perusteltiin. Sovellus ositettiin yhdeksi ydinosaksi ja useammaksi asiakaskohtaisiksi konfiguraatioksi. Konfiguraatioiden riippuvuudet kolmannen osapuolen kirjastoista ja muista hallinta-alkioista määriteltiin Apache Maven -projektinhallinta- ja kokoamistyökalun käyttämiin Project Object Model -tiedostoihin. Lopuksi jatkuvaan integraatioon ja kokoamisprosessiin tarvittavat työkalut asennettiin ja konfiguroitiin.

Insinöörityön tuloksena syntynyt prosessi osoittautui varsin hyödylliseksi, jonka ansiosta manuaalisen työn määrä väheni. Sovelluksen ydinosan ja asiakaskohtaisten konfiguraatioiden mahdolliset virheet voidaan löytää aikaisemmassa vaiheessa, ja niiden terveydentilaa voidaan seurata lähes reaaliajassa.

Jatkuvaan integraatioon siirtyminen ei ainoastaan tarkoittanut muutaman uuden työkalun käyttöönottoa, vaan se myös muutti kehitysprosessia ja työskentelytapaa kokonaisuudessaan. Jatkuva integraatio on erittäin tärkeä osa nykyaikaisia ketteriä ohjelmistokehitysmenetelmiä, eikä sen käyttöönotto useimmissa tapauksissa vaatine niin paljon aikaa, etteikö se maksaisi itseään kohtuullisessa ajassa takaisin.

Avainsanat	jatkuva integraatio, maven, jenkins, ketterät menetelmät

Metropolia

Abstract

Author(s) Title Number of Pages Date	Miku Laitinen Adoption of Continuous Integration 39 pages + 1 appendix 16 Apr 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Jari Eriksson, CTO Outi Grotenfelt, Senior Lecturer

The aim of this Bachelor's thesis was to design and adopt Continuous Integration for a Java web application developed by Playground Finland Ltd. The application was developed for collecting, managing and sharing information and it is currently being evaluated by a few clients. The application uses the Vaadin web application framework.

The complexity of the application and its development process began to increase as the number of clients and client-specific features increased. Due to the increased amount of manual effort and the resulting error proneness, a decision to adopt Continuous Integration was made.

The first step of this thesis was to evaluate and present the most appropriate tools for adopting Continuous Integration. The application was then split to a single core component and multiple client-specific configurations. Dependencies to third party Java libraries and other configuration objects were defined in Project Object Model files which are the core concept of Apache Maven project management and build tool. Finally, the tools needed for Continuous Integration and the build process were installed and configured.

As a result, a Continuous Integration process was successfully established. The process has proved to be worthwhile and the amount of manual effort required was reduced. Potential flaws in the core component and client-specific configurations can now be detected earlier and the health status of the application configurations can be monitored in near real-time.

The adoption of Continuous Integration was not only a matter of installing a number of new tools but also changing the software development process and working methods in their entirety. Continuous Integration has a crucial part in modern agile software development methods and in most cases adopting it is likely to return the financial investment within a reasonable period of time.

Keywords	continuous integration, maven, jenkins, agile methods

Sisällys

Lyhenteet ja määritelmät

CI	*Continuous Integration*. Jatkuva integraatio, ketterien ohjelmistokehitysmenetelmien käytäntö ohjelmiston automatisoituun ja usein tapahtuvaan kokoamiseen ja integrointiin.
JAR	*Java Archive.* Tiedostomuoto, jota voidaan käyttää Java-sovellusten ja -kirjastojen paketoimiseen.
POM	*Project Object Model.* Apache Maven -projektinhallintatyökalussa käytettävä projektin rakenteen määrittelevä XML-muotoinen tiedosto.
SSL	*Secure Sockets Layer.* Salausprotokolla, jolla voidaan suojata esimerkiksi yhteys web-sivuihin.
URL	*Uniform Resource Locator.* Protokollan ja yhteyden kohteen määrittävä merkkijono, esimerkiksi http://www.metropolia.fi.
WAR	*Web application Archive.* JAR:n kaltainen tiedostomuoto, joka on tarkoitettu erityisesti web-sovellusten paketoimiseen.
XML	*Extensible Markup Language.* Kuvauskieli, jota käytetään tiedon rakenteen ja sisällön kuvaamiseen.

1 Johdanto

Playground Finland Oy on kolmen mobiilialan uranuurtajan vuonna 2008 perustama suomalainen yritys, joka on keskittynyt mobiililaitteilla tapahtuvaan tiedonkeruuseen. Tämän insinöörityön tavoitteena on suunnitella ja toteuttaa eräälle Playground Finland Oy:n ohjelmistotuotteelle ketterien ohjelmistokehitysmenetelmien mukainen jatkuva integraatioprosessi (engl. Continuous Integration).

Työn lähtökohtana on muutamalla asiakkaalla testikäytössä oleva suomalaisen Vaadin-sovelluskehyksen päälle kehitetty Java-sovellus, jonka käyttötarkoituksena on tiedon kerääminen, hallinta sekä jakaminen. Sovelluksesta on tehty jokaisen asiakkaan käyttötarkoituksiin soveltuva versio, jossa sovelluksen ydinosa on kaikille sama, mutta osa komponenteista on asiakaskohtaisia. Jotkut näistä komponenteista muuttavat vastaavan ydinosan komponentin toimintaa ja jotkut tuovat lisää asiakkaan tarpeiden vaatimaa toiminnallisuutta. Asiakaskohtaiset uudet toiminnallisuudet on pyritty tekemään parametrisoimalla olemassa olevaa koodia siten, etteivät ne vaikuttaisi ydinosan ja täten myös muiden asiakaskohtaisten versioiden toimintaan.

Ongelmaksi on kuitenkin koitunut uusien asiakkaiden ja toiminnallisuuksien myötä kasvanut sovelluksen kompleksisuus, joka vaikeuttaa eri komponenttien riippuvuuksien hallintaa sekä lisää sovelluskehitysprosessin käyttöönottovaiheessa käsin tehtävää työtä. Tässä insinöörityössä pyritään ratkaisemaan edellä mainitut ongelmat selvittämällä ja tarvittaessa muokkaamalla komponenttien välisiä riippuvuuksia ja komponenttien riippuvuuksia kolmansien osapuolien kirjastoista, jakamalla ohjelmakoodin osat tarkemmin määriteltyihin komponentteihin sekä automatisoimalla käyttöönottovaiheessa tapahtuva käännös, integraatio ja käyttöönotto.

Työn ensimmäisessä vaiheessa valmistellaan sovellus jatkuvaan integraatioon paremmin soveltuvaksi selvittämällä tämänhetkiset komponenttien väliset riippuvuudet, muokkaamalla ja määrittelemällä komponentit selkeämmiksi sovelluksen osakokonaisuuksiksi sekä jakamalla määritellyt osakokonaisuudet versionhallinnassa omiin hakemistoihinsa. Työn toisessa vaiheessa asennetaan tarvittavat käännös-, integraatio- sekä riippuvuuksienhallintatyökalut ja konfiguroidaan ne toimimaan halutulla tavalla.

Luvussa 2 kerrotaan lyhyesti ja yleisluontoisesti ketteristä ohjelmistokehitysmenetelmistä ja niiden käyttämisestä saavutettavista eduista. Luku 2 antaa työn aiheelle perustan ja tiivistää lyhyesti sen, miksi ketterät ohjelmistokehitysmenetelmät ovat tänä päivänä niin suosittuja.

Luvussa 3 selitetään ohjelmistotuotteen hallintaa ja erilaisia metodeita, joita käyttämällä voidaan hillitä ohjelmistotuotteen kasvavaa kompleksisuutta ja hallita ohjelmistossa tapahtuvia muutoksia. Tämän luvun sisältämä teoria liittyy läheisesti lukuihin 5.2 ja 7.3, joissa asiaa käsitellään käytännönläheisemmin.

Seuraavassa luvussa 4 syvennytään jatkuvan integraatioprosessin toimintaan ja sen tuomiin hyötyihin. Jatkuva integraatio on tärkeä ketteriin menetelmiin kuuluva käytäntö, joka tässä insinöörityössä vaatii erityishuomiota.

Luvussa 5 esitellään erilaisia prosessin käyttöönottoa helpottavia sekä prosessiin käyttöönotettavia työkaluja. Alaluvut ovat siinä järjestyksessä, missä niitä työn aikana tullaan tarvitsemaan ja käyttämään.

Insinöörityön käytännön osuuden aloittaa luku 6, jossa tarkastellaan sovelluskehitysprojektin nykyistä tilaa ja käydään läpi jatkuvan integraatioprosessin käyttöönottoa varten vaadittavat muutokset lähdekoodissa ja komponenttirakenteessa. Tämä ja tätä edeltävä luku 5 ovat esivalmistelua jatkuvan integraatioprosessin luomista varten.

Luku 7 on koko insinöörityön ydin, jossa prosessin toteutus on kuvattu. Luvussa käsitellään myös lyhyesti jatkuvaa integrointiprosessia seuraavia vaiheita, jotka läpikäymällä sovellus saadaan kehitysympäristöstä tuotantokäyttöön.

2 Ketterä ohjelmistokehitys

2.1 Ketterien ohjelmistokehitysmenetelmien synty

80-luvulla ja 90-luvun alkupuolella yleinen näkemys ohjelmistoalalla oli, että parhaat tulokset ohjelmistotuotannossa saavutetaan huolellisella suunnittelulla, muodollisella laadunvarmistuksella, CASE-työkalujen tukemilla analyysi- ja suunnittelumetodeilla sekä hallitulla ja eksaktilla kehitysprosessilla. Tätä näkemystä tukivat eritoten suuria ja pitkäikäisiä järjestelmiä suunnitelleet IT-alan yritykset. Kun samanlaisia periaatteita sovellettiin myös pienempiä ohjelmistoprojekteja tekevissä pienissä ja keskisuurissa yrityksissä, käytettiin toisinaan määrittelyyn, suunnitteluun ja dokumentointiin merkittävästi enemmän aikaa kuin itse sovelluskehitykseen ja testaukseen. Kun ohjelmistotuotteen toiminnallisuutta muutettiin, täytyi määrittely- ja suunnitteludokumentit käydä uudestaan läpi ja päivittää ajan tasalle. [Sommerville 2007: 396.]

Tyytymättömänä määrittelyn, suunnittelun ja dokumentoinnin runsaaseen määrään ryhmä ohjelmistokehittäjiä kehitti 90-luvulla niin sanotut ketterät ohjelmistokehitysmenetelmät (engl. Agile Software Development), joissa kehitys tapahtuu pienissä, yleensä dokumentoinnista ja sovelluskehityksestä koostuvissa, sykleissä. Näiden uusien menetelmien ansiosta ohjelmistokehittäjät pystyivät keskittymään itse ohjelmistotuotteen kehittämiseen sen suunnittelun ja alituisen dokumentoinnin sijaan. [Sommerville 2007: 396; Iteratiivinen ja inkrementaalinen kehitys 2008.]

2.2 Pääperiaatteet

Seuraava lainaus on ketterien menetelmien kehittäjien vuonna 2001 laatima manifesti (Manifesto for Agile Software Development), jossa on lyhyesti selitetty ketterien menetelmien kantava ajatus:

> Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä
> itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan:
> **Yksilöitä ja vuorovaikutusta** enemmän kuin prosesseja ja työkaluja
> **Toimivaa sovellusta** enemmän kuin kokonaisvaltaista dokumentaatiota
> **Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluita
> **Muutokseen reagoimista** enemmän kuin suunnitelman noudattamista.
> Vaikka oikealla puolellakin on arvoa, arvostamme vasemmalla puolella olevia asioita enemmän. [Ketterät käytännöt 2008.]

Ketterät menetelmät ovat luonteeltaan keskenään samankaltaisia ja tähtäävät samaan lopputulokseen toteuttaen samoja periaatteita, mutta ne ovat kuitenkin sisällöltään toisistaan enemmän tai vähemmän poikkeavia. Yhteisiä tekijöitä menetelmille ovat asiakkaan tiivis osallistuminen kehitysprosessiin, ohjelmiston kehitys ja toimittaminen inkrementeissä, kehittäjälle annettavat vapaat kädet työtapojen valintaan, muutoksien omaksuminen ja niihin varautuminen sekä ohjelmiston ja ohjelmistokehitysprosessin yksinkertaisuuteen keskittyminen. [Sommerville 2008: 396–397.]

Ketteriä ohjelmistokehitysmenetelmiä käyttämällä voidaan saavuttaa tiettyjä etuja verrattuna perinteiseen ohjelmistokehitykseen. Näitä etuja ovat muun muassa:

- Riskit havaitaan aikaisemmin.
- Muutokset ovat paremmin hallittavissa.
- Ohjelmistokehittäjät voivat keskittyä enemmän itse ohjelmistoon.
- Ohjelmistokehittäjillä paremmat mahdollisuudet oppia uutta.
- Asiakas pääsee vaikuttamaan enemmän kehitysprosessiin.
- Asiakas pääsee kokeilemaan työn tuloksia aikaisemmin. [Vuori 2009: 24; Sommerville 2008: 396–397.]

On kuitenkin otettava huomioon, etteivät ketterät ohjelmistokehitysmenetelmät välttämättä sovellu kaikenlaisten järjestelmien kehittämiseen. Sommerville [2008: 398] ei usko ketterien menetelmien soveltuvan niin hyvin suurten tai kriittisten järjestelmien kuin pienten ja keskisuurten yritysjärjestelmien kehittämiseen.

2.3 Inkrementaalinen ja iteratiivinen kehitys

Inkrementaalinen kehitys tarkoittaa ohjelmistotuotteen toimittamista pienissä ohjelmistoa täydentävissä osissa eli pienissä julkaisuissa. Inkrementin on oltava laadultaan asennuskelpoinen ja suotavaa olisikin asentaa se sen lopulliseen käyttöympäristöönsä nopean käyttäjäpalautteen saamiseksi. Yksi inkrementti kestää yleensä 1–6 viikkoa ja se toteutetaan iteraatioissa, jotka myös kestävät ennalta määrätyn lyhyehkön ajanjakson. [Iteratiivinen ja inkrementaalinen kehitys 2008; Pienet julkaisut 2008.]

Iteraatio tarkoittaa jonkin asian toistamista uudelleen ja uudelleen. Tässä yhteydessä sillä tarkoitetaan toistuvia ajanjaksoja, joiden aikana inkrementtiä kehitetään. Iteraation lopuksi asiakas voi vahvistaa iteraatioissa toteutettujen osien toiminnan ja tarvittaessa täsmentää vaatimuksia. Seuraavan iteraation jälkeen asiakas voi jälleen vahvistaa aiemmin täsmentämiensä vaatimusten tuottamat tulokset ja täten osallistua hyvin läheisesti ohjelmistotuotteen kehittämiseen. [Iteratiivinen ja inkrementaalinen kehitys 2008.]

3 Konfiguraationhallinta

3.1 Konfiguraationhallinta ja sen tuomat hyödyt

Konfiguraationhallinta, jota kutsutaan toisinaan myös nimellä ohjelmistotuotteen hallinta, tarkoittaa ohjelmistotuotteen kokonaisvaltaista hallintaa sen elinkaaren aikana. Se on kriittinen osa ohjelmistotuotantoa ja sitä tarvitaan ohjelmistojen kasvaneen kompleksisuuden, kasvaneen kysynnän ja muutosherkän luonteen vuoksi. [Koskela 2003: 9.]

Konfiguraationhallinta ei ole yksi ennalta määrätty menetelmä, vaan sen luonne ja osuus ohjelmistokehitystyössä määräytyy hyvin pitkälti organisaation toimintamallien ja projektin luonteen mukaan. Suurissa projekteissa konfiguraationhallinta voi olla niin merkittävässä osassa, että se työllistää yhden tai useamman henkilön täysipäiväisesti. [Haikala & Märijärvi 2006: 256.]

Tehokkaasti käytettynä konfiguraationhallinta voi olla merkittävä kilpailuetu. Ohjelmistotuotteen koko elinkaaren aikana käytettynä konfiguraationhallinnalla voidaan tunnistaa kehitettävät hallinta-alkiot, helpottaa ja selkeyttää muutostenhallintaa, saada tarvittaessa tietoa kehityksen edistymisestä ja helpottaa ohjelmistotuotteen sekä konfiguraationhallintaprosessin auditointia. Tarkoituksena on siis tukea ohjelmistokehitystä ja parantaa ohjelmiston laatua. [Koskela 2003: 10.]

3.2 Peruskäsitteet

Komponentti

Ohjelmistotuotteen katsotaan usein koostuvan komponenteista, joita ovat muun muassa ohjelman lähdekoodi, lähdekoodista käännetyt ohjelmatiedostot, kääntämiseen tarvittavat skriptit, testidata, työkalut ja määrittely-, suunnittelu- sekä testausdokumentit. Konfiguraationhallinnassa komponentin oleelliset attribuutit ovat sen yksityiskohtainen kuvaus, versio, tiedot ajan saatossa tehdyistä muutoksista, käännösohjeet sekä status (esimerkiksi testaamaton, yksikkötestattu). [Haikala & Märijärvi 2006: 255-257; Koskela 2003: 11.]

Komponentti voi myös olla johdettu muista komponenteista. Esimerkkinä mainittakoon lähdekoodista kääntäjällä käännetyt tietokoneen tai virtuaalikoneen ymmärtämät binääritiedostot. [Haikala & Märijärvi 2006: 257.]

Konfiguraatio

Komponenttiin ja konfiguraatioon voidaan molempiin viitata yleistäen nimellä *hallinta-alkio*. Konfiguraatio koostuu hallinta-alkioista, ja se voi muodostaa kokonaisen ohjelmistotuotteen tai osan tuotteesta. Komponentin tapaan myös konfiguraatiot versioidaan, ja jonkin konfiguraation hallinta-alkion tai sen version muuttuessa myös konfiguraation versio muuttuu. [Haikala & Märijärvi 2006: 257–258.]

Versio

Versio tarkoittaa jäädytettyä hallinta-alkiota. Jäädyttämisellä tarkoitetaan tässä yhteydessä hallinta-alkion asettamista tilaan, jossa siihen ei enää voi tehdä muutoksia. Uusia muutoksia varten on luotava uusi versio. [Haikala & Märijärvi 2006: 257.]

Vaihetaso

Vaihetasolla (engl. baseline) tarkoitetaan ohjelmistokehitysprojektin aikana hallinta-alkion viimeisintä versiota. Jokaisella komponentilla ja konfiguraatiolla on siis vaihetaso.

Konfiguraatio ei kuitenkaan välttämättä koostu vaihetasoista, vaan se voi sisältää vanhempia versioita hallinta-alkioista. [Haikala & Märijärvi 2006: 259–260.]

3.3 Versionhallinta

Versionhallinnan tarkoituksena on hallita ohjelmistotuotantoprosessin aikana luotujen tiedostojen eri versioita. Tiedostojen versioiden hallinta manuaalisesti on erittäin virhealtista ja sitä varten on kehitetty useita versionhallintatyökaluja, joiden tehtävä on automatisoida versiointiprosessia ja vähentää virhealttiutta. Versionhallintatyökalut pitävät kirjaa projektin ja sen tiedostojen versiohistoriasta, josta näkee, kuka on tehnyt muutoksia, mitä muutoksia on tehty, milloin ne on tehty ja miksi ne on tehty. Mikään muutos ei ole versionhallintatyökaluja käytettäessä peruuttamaton, eli historiaan on aina mahdollista palata. [Koskela 2003: 18; O'Sullivan 2009: 1–2.]

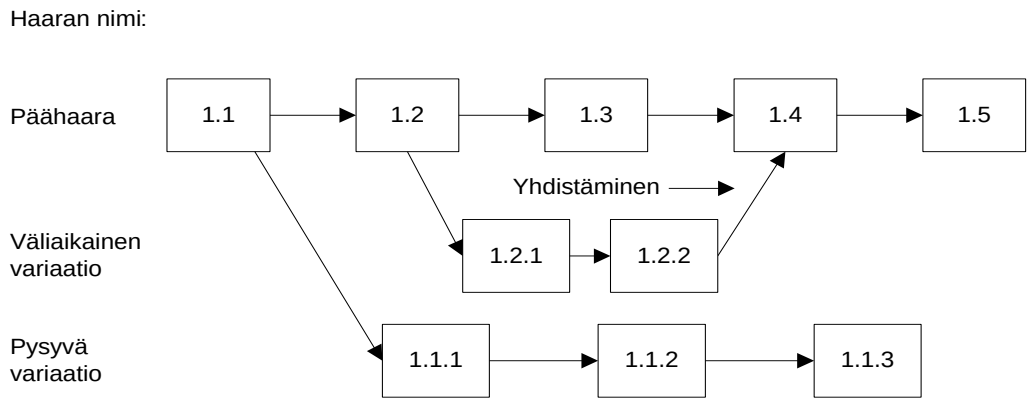
Versionhallintatyökalut helpottavat yhteistyötä muiden kehittäjien kanssa. Jos kehittäjät tekevät keskenään epäyhteensopivia muutoksia ohjelmistoon, voidaan käytössä olevan versionhallintatyökalun avulla tunnistaa ja ratkaista epäyhteensopivien osien aiheuttamat konfliktit. [O'Sullivan 2009: 2.]

Niin kauan kuin hallinta-alkio ei ole konfiguraationhallinnan alaisuudessa, voi kehittäjä vapaasti tehdä siihen muutoksia. Kun hallinta-alkio lisätään konfiguraationhallinnan piiriin, se versioidaan ja siihen tehtävät muutokset tuottavat *revisioita* (engl. revision). Revisiolla tarkoitetaan lineaarisesti kasvavassa versiopuussa peräkkäistä versiota, eli ensimmäisen versionumeron ollessa 1.0, on seuraava versionumero eli revisio 1.1. Uusi revisio luodaan yleensä virheitä korjattaessa tai uutta toiminnallisuutta lisättäessä. [Haikala & Märijärvi 2006: 260–261; Koskela 2003: 18.]

Jos versiopuun päähaarasta luodaan sivuhaara, kutsutaan *haaran* (engl. branch) versiota *variaatioksi* (engl. variant). Haaroittaminen ei ole aina välttämätöntä variaation luomiseksi, vaan käytettävän version voi ratkaista myös parametritiedostoilla, ehdollisella lähdekoodin kääntämisellä (esimerkiksi C-kielessä #ifdef-rakenne) tai tarkistuksella ohjelmiston suorituksen aikana. Variaatioita on olemassa kahdenlaisia: väliaikaisia ja pysyviä. Pysyvät variaatiot ovat esimerkiksi eri alustalle suunnattuja tai asiakkaan tarpeisiin paremmin soveltuvia versioita. Väliaikaiset variaatiot ovat nimensä mukaisesti

väliaikaisia, ja ne on tarkoitus yhdistää myöhemmin ohjelmistokehitysprojektin aikana toiseen variaatioon tai päähaaraan. [Haikala & Märijärvi 2006: 261–264; Koskela 2003: 18.]

Kuviossa 1 on esimerkki versiopuun jakautumista päähaaraan, väliaikaiseen sivuhaaraan ja pysyvään sivuhaaraan.



Kuvio 1. Versiopuun haarat [Koskela 2003: 19].

3.4 Konfiguraatiot ja muutostenhallinta

Konfiguraationhallinnassa on asiakaskohtaisille konfiguraatioille määritetty niiden sisältämät hallinta-alkiot sekä ohjeet niiden toimintakuntoon saattamiseen. Ohjelmistotuotteen konfiguraatioon tehdään muutoksia silloin kun vaatimukset muuttuvat, kun tarvitaan uutta toiminnallisuutta tai kun ohjelmistosta on löytynyt virhe. Ennen muutosten tekemistä on kuitenkin selvitettävä, ovatko muutospyynnöt ja tehtävät muutokset täysin aiheellisia. Hylättäväksi päätyvän muutospyynnön taustalla saattaa olla ohjelmiston toimintaan liittyvä väärinkäsitys tai duplikaatti muutospyyntö. [Sommerville 2008: 695–697; Haikala & Märijärvi 2006: 263–264.]

Muutospyynnön hyväksymisen jälkeen on otettava selvää, mihin komponenttiin tai komponentteihin ja mihin versioihin muutokset on tehtävä. Muutoksia suunniteltaessa on huomioitava ja identifioitava myös muut asiakkaat, joiden konfiguraatioon muutokset vaikuttavat. [Haikala & Märijärvi 2006: 264.]

4 Jatkuva integraatio

4.1 Kokoaminen

Termillä kokoaminen (engl. build) tarkoitetaan yleensä muutakin kuin pelkkää lähdekoodin kääntämistä (engl. compile). Kokoamiseen voi sisältyä kääntämisen lisäksi esimerkiksi testien ajaminen, lähdekoodin staattinen ja/tai dynaaminen analyysi sekä kokoamisprosessin lopputuloksen käyttöönotto. Kokoamisen tarkoituksena on siis koota ohjelman osat yhteen ja varmistaa, että ohjelma toimii kokonaisuudessaan. [Duvall ym. 2007: 4.]

Kokoamisprosessin automatisointi vähentää manuaalisen, itseään toistavan ja virhealttiin työn määrää. Kokoamisprosessi voidaan automatisoida kirjoittamalla kokoamisskriptejä (engl. build script), joita voidaan suorittaa käyttämällä erilaisia kokoamistyökaluja. Automaattisen kokoamisprosessin tulisi olla niin kattava, että yhdellä kokoamistyökalulle annettavalla komennolla saisi ohjelmiston koottua ja toimintakuntoon. [Duvall ym. 2007: 67–68.]

Java on osittain tulkattu ohjelmointikieli ja se eroaa perinteisistä staattisesti käännettävistä ohjelmointikielistä (esimerkiksi C/C++) muun muassa siten, ettei sitä käännösvaiheessa käännetä suoraan tietokoneen suorittimen ymmärtämälle konekielelle. Sen sijaan se käännetään tavukoodiksi (engl. bytecode), joka suoritetaan Java-virtuaalikoneessa (engl. Java Virtual Machine). Tavukoodi on ikään kuin konekieltä Java-virtuaalikoneelle, joka ohjelman suorituksen aikana tulkkaa sen järjestelmäkohtaiselle konekielelle. [Kosonen ym. 2007: 12; Perry & Oskov 2004.]

4.2 Jatkuvan integraation vaiheet

Nimestään huolimatta jatkuva integraatio ei ole oikeastaan jatkuvaa (engl. continuous), vaan ennemminkin usein toistuvaa (engl. continual). Sananmukaisesti jatkuva tarkoittaisi sellaista prosessia, joka kerran käynnistyisi eikä sen koommin pysähtyisi. [Duvall ym. 2007: xxii.]

CI-palvelin (engl. Continuous Integration Server, Integration Build Machine) on jatkuvaan integraatioon tarkoitettu palvelin, jonka ensisijainen tehtävä on seurata *versionhallintavarastoon* (engl. repository) tehtäviä muutoksia ja reagoida niihin noutamalla muuttuneet tiedostot sekä suorittamalla kokoamisen. Versionhallintavaraston voi konfiguroida ilmoittamaan CI-palvelimelle muutoksista heti kun ne on tehty, mutta toisinaan voi olla suotavampaa antaa CI-palvelimen seurata versionhallintavarastoon tehtäviä muutoksia säännöllisin väliajoin, esimerkiksi muutaman minuutin välein. [Duvall ym. 2007: 8 & 12.]

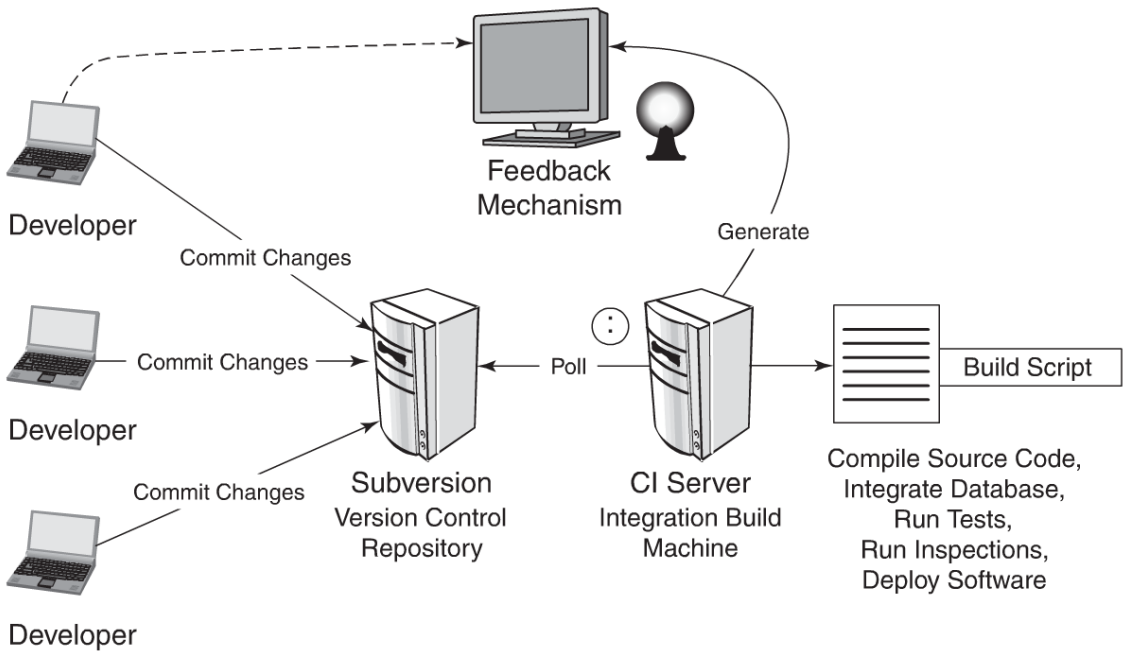
Ennen kuin ohjelmistotuotteen kehittäjä tallentaa (engl. commit changes) lähdekoodiin tai muuhun ohjelmiston hallinta-alkioon tekemänsä muutokset versionhallintavarastoon, on integraatiokokoamisen (engl. integration build) epäonnistumisen ehkäisemiseksi suotavaa suorittaa niin sanottu yksityinen kokoaminen (engl. private build) omassa kehitysympäristössä ja varmistaa, ettei virheitä esiinny. Kun yksityinen kokoaminen on suoritettu ja kaikki yksikkötestit on läpäisty, voidaan muutokset viedä versionhallintavarastoon. Mikäli joku muu on ennättänyt viedä tekemänsä muutokset versionhallintavarastoon ensin, on toisen kehittäjän muutokset haettava versionhallinnasta omaan työkopioon (engl. working copy), jonka jälkeen muutokset voidaan viedä versionhallintaan. [Duvall ym. 2007: 5 & 42 & 79.]

Versionhallintavarastoon tallennettujen muutosten jälkeen CI-palvelin noutaa sieltä viimeisimmän revision ja suorittaa kokoamisskriptin, joka kokoaa ohjelmiston [Duvall ym. 2007: 8]. Kuten luvussa 4.1 mainittiin, kokoaminen voi sisältää myös testien ajamisen sekä staattisen ja/tai dynaamisen analyysin. Näiden lisäksi se voi ottaa juuri kootun ohjelmiston testipalvelimelle käyttöön, mikäli näin on kokoamisskriptissä määritetty. [Fowler 2006.]

Kun ohjelmistosta halutaan tehdä julkaisuversio ja ottaa se tuotantokäyttöön, voidaan CI-palvelimella suorittaa kokoamisskripti, joka tekee kaikki käyttöönottoon vaadittavat toimenpiteet. Julkaisuversion kokoamisskripti voi esimerkiksi noutaa halutun version versionhallintavarastosta, merkitä version julkaisuversioksi versionhallintavarastoon, koota ohjelmiston, merkitä kootun ohjelmiston julkaisuversioon pohjautuvalla merkinnällä ja ottaa ohjelmiston käyttöön tuotantopalvelimella. Tarpeen vaatiessa julkaisuver-

sio voidaan palauttaa edelliseen versioon käyttäen versionhallintavarastoon merkittyjä julkaisuversioita. [Duvall ym. 2007: 191-199.]

Jatkuva integraatioprosessi on kuvattuna kuviossa 2. Integraatio käynnistyy, kun joku kehittäjistä tekee versionhallintavarastoon muutoksia, minkä jälkeen CI-palvelin noutaa viimeisimmän revision versionhallintavarastosta, kokoaa sen ja luo raportin kokoamisen onnistumisesta. Lopuksi kehittäjät voivat tarkastella kokoamisen tuloksia palautemekanismin avulla, joka voi olla esimerkiksi raportin web-sivu, sähköpostiviesti tai jopa tekstiviesti.



Kuvio 2. Jatkuva integraatioprosessi [Duvall ym. 2007: 5].

4.3 Jatkuvan integraation tuomat hyödyt

Fowler [2006] uskoo, että suurin jatkuvan integraation tuoma hyöty on vähentyneet riskit. Duvall ym. [2007: 29] myötäilee tätä ja sijoittaa vähentyneet riskit jatkuvan integraation tuomien hyötyjen listalle ensimmäiseksi.

Riskit vähenevät muun muassa silloin, kun virheet ohjelmassa huomataan aikaisemmassa vaiheessa. Aikaisin löydetyt virheet ovat yleensä helpompia korjata kuin myöhemmin löydetyt, sillä virhe on tällöin tuoreessa muistissa ja se löytyy todennäköisim-

min äskettäin versionhallintavarastoon viedyistä muutoksista. Toinen riskejä vähentävä tekijä on kyky mitata ohjelmiston "terveydentilaa". Automaattisilla testeillä ja analyyseilla voidaan kokoamisen yhteydessä seurata ohjelmiston laatua ja kompleksisuutta. [Duvall ym. 2007: 29–30; Fowler 2006.]

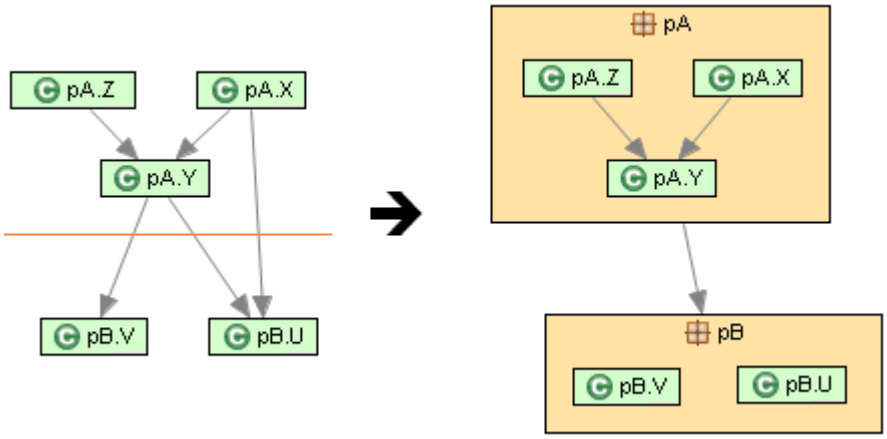
Jatkuva integraatio vähentää myös niin sanottua liukuhihnatyötä ja vapauttaa henkilöresursseja tärkeämpiin, ajatusta vaativiin, työtehtäviin. Prosessia ei tarvitse käynnistää manuaalisesti ja raportoinnin voi toteuttaa siten, ettei kehittäjien tarvitse itse erikseen ottaa selvää kokoamisen onnistumisesta. [Duvall ym. 2007: 30–31.]

5 Käytettävät työkalut

5.1 Ispace

Ispace on yksinkertainen ja joustava Eclipse-sovelluskehitysympäristön lisäosa, jolla voi visualisoida, analysoida ja kokeellisesti järjestellä uudelleen Java-projektien riippuvuuksia. Ispace käyttää tuottamissaan kaavioissa sisäkkäisiä rakenteita hierarkkisen tiedon esittämiseen. [Aracic 2010; Aracic 2008.]

Kuviossa 3 on esimerkki luokkahierarkioiden esitystavasta Ispacessa. Esimerkissä kaikki kolme relaatiota pakkauksen pA ja pB välillä on ryhmitetty yhdeksi relaatioksi pakkausten pA ja pB välille [Aracic 2008].



Kuvio 3. Relaatioiden ryhmittely [Aracic 2008].

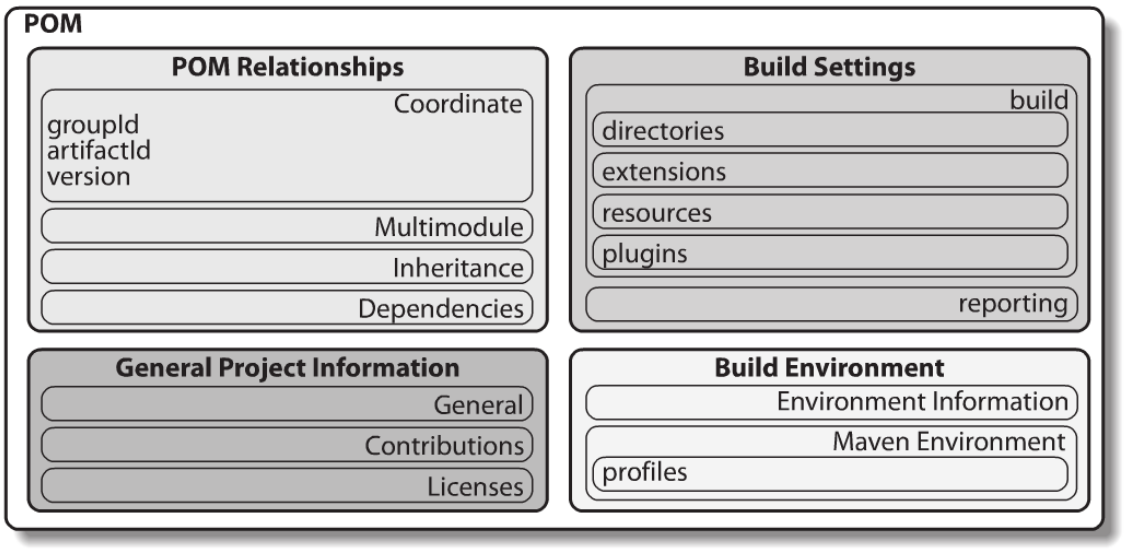
Ispacea tarvitaan tässä insinöörityössä sovelluksen pakkausten ja luokkien riippuvuuksien kartoittamiseen, sillä ilman riippuvuuksien tuntemista on vaikeaa paloitella projekti pienemmiksi osakokonaisuuksiksi. Työkalua ei todennäköisesti tarvita enää myöhemmissä vaiheissa. Komponenttien riippuvuuksien tunnistaminen Ispacen avulla on selostettu luvussa 6.2.

5.2 Apache Maven

5.2.1 Maven ja Project Object Model

Maven on projektinhallinta- ja kokoamistyökalu, jonka kantavana ajatuksena on alusta asti ollut yksinkertaistaa ja yhdenmukaistaa projektien rakennetta sekä kokoamisprosessia [Sonatype 2008: xi]. Apache Software Foundation alkoi kehittää Mavenia tarkoituksenaan yksinkertaistaa Jakarta Turbine -projektin kokoamisprosesseja [What Is Maven? 2011].

Mavenin keskeisin käsite on projektin rakennetta ja kokoamisprosessia kuvaava Project Object Model (lyhenne POM). POM on XML-muotoinen tiedosto, joka on sijoitettava projektin juurihakemistoon nimellä `pom.xml`. POM-tiedoston sisältävää projektia kutsutaan Mavenissa nimellä *artefakti*. POM-tiedoston kuvaustiedot ja konfiguraatiot on jaettu neljään kategoriaan, joita ovat yleiset projektitiedot, kokoamisasetukset, kokoamisympäristö sekä riippuvuudet. Yleisissä projektitiedoissa voidaan määritellä projektin nimi, projektin kotisivujen osoite, sponsoroiva organisaatio, lista projektiin osallistuneista henkilöistä sekä projektin lisenssi. Kokoamisasetuksissa voidaan konfiguroida Mavenin määrittelemiä konventioita kokoamisen osalta, esimerkiksi vaihtaa lähdekoodin sijaintia tai lisätä kokoamisprosessiin erilaisia toimenpiteitä suorittavia liitännäisiä. Kokoamisympäristöissä määritellään tarpeelliset ympäristöt, joissa projekti tullaan kokoamaan. Näitä voivat olla esimerkiksi kehitys- ja tuotantoympäristö. Neljäs kategoria, yhteydet, sisältää POM:n ainoat pakolliset tiedot, joita ovat ryhmätunniste, artefaktitunniste sekä versio. Näiden lisäksi se voi sisältää riippuvuudet muihin artefakteihin, mikäli riippuvuuksia on. Kuviossa 4 on edellisiin kuvauksiin liittyvä POM-tiedoston rakenne. [Sonatype 2008: 6 & 149–151.]



Kuvio 4. POM-tiedoston rakenne [Sonatype 2008: 150].

Yksi Mavenin keskeisistä periaatteista on suosia konventioita konfiguroinnin sijaan. Aikaa vievän konfiguroinnin sijaan voidaan noudattaa konventiota, eli Mavenin määrittelemiä suosituksia ja käytäntöjä projektin rakenteesta sekä projektin kokoamisprosessista. Maven käyttää siis asetusmäärittelyissä oletusarvoja, joita voi tarpeen vaatiessa konfiguroida omiin tarpeisiin. Maven olettaa lähdekoodin löytyvän hakemistosta `${juuri}/src/main/java`, resurssitiedostojen hakemistosta `${juuri}/src/main/resources` ja testitiedostojen hakemistosta `${juuri}/src/test`, joissa `${juuri}` on POM-tiedoston sisältävä projektin juurihakemisto. Hakemistot voivat sijaita muuallakin, mutta tällöin niiden sijainti on konfiguroitava POM-tiedostoon. [Sonatype 2008: 3.]

5.2.2 Kokoamisprosessi

Kuten edellisessä luvussa mainittiin, on Mavenilla omat konventionsa kokoamisprosessillekin, jota kutsutaan Mavenissa nimellä *build lifecycle*. Kokoamisprosessi koostuu *vaiheista*, jotka ovat ennalta määritettyjä, mutta tarvittaessa muokattavissa.

Mavenin oletus-lifecyclen vaiheita ovat

- `validate`, joka tarkistaa, että POM on validi ja että kaikki oleellinen tieto on saatavilla.
- `compile`, joka kääntää projektin lähdekoodin.

- `test`, joka suorittaa yksikkötestit äskettäin käännetylle lähdekoodille.
- `package`, joka paketoi käännetyn lähdekoodin annettuun tiedostomuotoon (oletuksena JAR).
- `integration-test`, joka käsittelee ja ottaa äskettäin paketoidun ohjelmiston käyttöön ympäristössä, jossa integraatiotestit voidaan suorittaa.
- `verify`, joka tarkistaa, että paketti on validi ja täyttää sille asetetut laatuvaatimukset.
- `install`, joka kopioi paketin paikalliseen artefaktivarastoon ja täten sallii äskettäin paketoidun projektin käyttämisen muiden paikallisten projektien riippuvuutena.
- `deploy`, joka kopioi paketin ulkoiseen artefaktivarastoon (esimerkiksi yrityksen tai projektin) muiden kehittäjien ja projektien käytettäväksi.

Vaiheet suoritetaan edellä luetellussa järjestyksessä. Vaiheita ei kuitenkaan tarvitse luetella Mavenille manuaalisesti, vaan esimerkiksi komento `mvn deploy` suorittaa kaikki `deploy`-vaihetta edeltävät vaiheet ja lopuksi komennossa määritellyn vaiheen, eli `deploy`-vaiheen. [Sonatype 2008: 181; Introduction to the Build Lifecycle 2011.]

Kokoamisprosessin vaihe koostuu yhdestä tai useammasta *maalista* (engl. goal), joka edustaa yksittäistä kokoamista edistävää tehtävää. Jos vaiheessa ei ole yhtäkään maalia, sitä ei suoriteta. Sama maali voi esiintyä yhdessä tai useammassa vaiheessa tai sitten ei yhdessäkään. Maali, joka ei kuulu yhteenkään vaiheeseen, voidaan siitä huolimatta suorittaa suoraan komentoriviltä samaan tapaan kuin vaiheetkin. [Introduction to the Build Lifecycle 2011.]

Mavenin ydin on rakenteeltaan hyvin yksinkertainen, eikä se itsessään kykene tekemään kovinkaan montaa asiaa. Maven on suunniteltu delegoimaan kaikki tärkeät tehtävät *liitännäisille.* Täten kaikki Mavenin ydin- sekä lisätoiminnot suoritetaankin liitännäisissä ja niiden määrittelemissä maaleissa. Tästä on käytännön hyötyä muun muassa silloin, kun jostakin liitännäisestä julkaistaan uusi versio. Tällöin koko Mavenia ei tarvitse päivittää, vaan pelkän liitännäisen päivittäminen riittää. Liitännäisen päivittämiseen ei vaadita muuta kuin sen versionumeron päivittäminen POM-tiedostoon. [Sonatype 2008: 5.]

Eräs tässä projektissa tarvittava liitännäinen on WAR plugin eli WAR-liitännäinen. WAR-liitännäisen tehtävänä on paketoida web-sovelluksen kokoamisprosessin lopputulos WAR-muotoiseen tiedostoon, joka sisältää sovelluksen suorittamiseen tarvittavat tiedostot. WAR-liitännäistä ei tarvitse erikseen konfiguroida POM-tiedostoon, vaan se otetaan automaattisesti käyttöön silloin, kun koottavan artefaktin tyypiksi on määritetty WAR. Projektin kannalta tärkeä osa WAR-liitännäistä on *overlay*-toiminto, jota käytetään resurssien jakamiseen useamman web-sovelluksen kesken. Overlay-toimintoa käytettäessä web-sovellukset yhdistetään niin sanotulla first-win -periaatteella, jonka mukaan web-sovelluksesta kopioitua tiedostoa ei voida ylikirjoittaa toisen web-sovelluksen vastaavalla tiedostolla. Jos esimerkiksi sovelluksissa A ja B on samoissa hakemistoissa tiedosto nimeltä `web.xml` B:n ollessa A:n riippuvuus, käytetään sovelluksen A `web.xml`-tiedostoa. Overlay-toimintoa ei tarvitse erikseen konfiguroida, vaan se on automaattisesti käytössä mikäli artefaktin riippuvuus on tyypiltään WAR. Overlay voidaan konfiguroida sisällyttämään tai sulkemaan pois haluttuja tiedostoja, mutta tässä projektissa overlayn konfiguroinnille ei ole tarvetta. [Maven WAR Plugin 2010; Overlays 2010.]

5.2.3 Mavenin valintakriteerit

Maven ei suinkaan ole ainoa työkalu, jolla kokoamisprosessia voidaan automatisoida ja ottaa osaksi jatkuvaa integraatiota. Toinen varteenotettava vaihtoehto on Apache Ant, joka keskittyy lähinnä vain kokoamiseen.

Ant ei määrittele konventioita Mavenin tapaan, vaan kaikki hakemistojen sijaintia ja kokoamisprosessin kulkua myöten on konfiguroitava manuaalisesti kokoamistiedostoon. Antia voidaan tarvittaessa käyttää osana Mavenin kokoamisprosessia ja tällainen menettely onkin kohdallaan silloin, kun kokoaminen sisältää monimutkaisia ja Mavenin konventioihin soveltumattomia vaiheita. Ant sopii siis parhaiten monimutkaisiin kokoamisprosesseihin. [Sonatype 2008: 9–10.]

Yksi merkittävä tekijä valinnassa on sovelluksessa käytettävän Vaadin-sovelluskehyksen hyvä soveltuminen Maven-ympäristöön. Vaadin määrittelee oman Maven-projektityyppinsä, joka muun muassa sisältää yhden käännösvaiheen lisää kokoamisprosessiin. Edellä mainittua käännösvaihetta tarvitaan Java-lähdekoodin kääntämiseksi

HTML- sekä JavaScript-koodiksi tapauksissa, joissa projekti sisältää web-sovelluksen selainpuolella suoritettavia lisäosia. [Using Vaadin with Maven 2010.]

Sovelluksessa käytössä olevat Vaadin-sovelluskehyksen lisäosat ja kaikki muut kolmannen osapuolen kirjastot sijaitsevat joko Vaadinin artefaktivarastossa tai Mavenin keskusartefaktivarastossa. Tämä helpottaa merkittävästi projektin konfiguraatioiden sisältämien eri hallinta-alkioiden, kuten asiakaskohtaisten komponenttien, kolmannen osapuolen kirjastojen sekä Vaadin-lisäosien versioiden hallintaa eli konfiguraationhallintaa.

5.3 Sonatype Nexus

Sonatype Nexus on Mavenin artefaktivarastojen hallintaan tarkoitettu työkalu. Artefaktivarastonhallintatyökalut toimivat välityspalvelimina Mavenin keskusartefaktivarastojen ja yrityksen tai muun organisaation artefaktivarastojen välillä. Tämän lisäksi ne tarjoavat säilytyspaikan organisaation sisällä toteutetuille artefakteille. [Sonatype 2008: 333.]

Artefaktivarastonhallintatyökaluilla voidaan suodattaa ja valikoida organisaatiossa käytettävät artefaktit. Organisaatio voi esimerkiksi rajata pois sellaiset artefaktit, joiden lisenssit eivät ole yhteensopivia kehitettävän ohjelmiston lisenssin kanssa. Organisaatio voi myös tarvittaessa käyttää artefaktista vain tiettyä versiota ja estää pääsyn artefaktin muihin versioihin. [Sonatype 2008: 334.]

Välityspalvelimina organisaation sisällä toimiessaan artefaktivarastonhallintatyökalut tallentavat kehittäjien niiden kautta lataamat artefaktit omaan sisäiseen artefaktivarastoonsa ja täten nopeuttavat usein tarvittavien artefaktien lataamista. Käytännössä siis tarpeelliset artefaktit haetaan julkisesta artefaktivarastosta vain ensimmäisellä latauskerralla ja seuraavilla latauskerroilla, esimerkiksi muiden kehittäjien tarvitessa kertaalleen ladattuja artefakteja, ne haetaan organisaation sisäisestä artefaktivarastosta. [Sonatype 2008: 333.]

Sonatype Nexus valittiin käyttöönotettavaksi artefaktivarastonhallintatyökaluksi muun muassa seuraavien vahvuuksien johdosta:

- Tiheä julkaisuväli (uusi julkaisuversio joka 6. viikko)
- hyvä yhteensopivuus Sonatype m2eclipsen kanssa (esitelty luvussa 5.5)

- erittäin tehokas muistinkäyttö
- sisäänrakennettu julkisten artefaktivarastojen selailutoiminto
- helppokäyttöisyys.

Muita varteenotettavia vaihtoehtoja olivat Apache Archiva ja JFrogin kehittämä Artifactory. Edellisistä jälkimmäiselle on saatavilla maksullinen tehokäyttöön tarkoitettu Pro-versio, kuin myös Nexukselleki. [Maven Repository Manager Feature Matrix 2011.]

5.4 Jenkins

Jenkins on Sun Microsystemsillä (nykyään Oracle) työskennelleen Kohsuke Kawaguchin kehittämä web-pohjainen jatkuvaan integraation käytettävä sovellus, joka monitoroi toistuvasti suoritettavien tehtävien, kuten ohjelmistojen kokoamisen, suoritusta. Sillä voidaan seurata versionhallinnassa tapahtuvia muutoksia ja käynnistää kokoaminen versionhallinnasta noudettujen muutosten pohjalta. Jenkins on lisensoitu avoimen lähdekoodin MIT-lisenssillä ja täten myös ilmainen. [Meet Jenkins 2011; Wiest 2010.]

Jenkins oli ennen 29.1.2011 nimeltään Hudson. Hudson-projektin kehittäjä Andrew Bayer [2011a] kirjoitti tammikuussa 2011 Hudsonin tulevaisuudesta ja totesi, että projektin nimi tulee todennäköisesti vaihtumaan. Syyksi hän mainitsee sen, että ostettuaan Sun Microsystemsin Oracle sai omistusoikeuden Hudsonin tavaramerkkeihin Euroopan unionissa sekä Amerikan yhdysvalloissa, eikä Hudson-yhteisö halunnut Oraclen sääntelevän kehitystä. Projektin toiminta tulee jatkumaan entiseen tapaan, eikä kehityksen jatkuminen ole vaarassa tämän takia. [Bayer 2011a; Bayer 2011b.]

Jenkinsille on olemassa vaihtoehtoja, mutta mikään niistä ei sovi tähän projektiin yhtä hyvin kuin Jenkins. Jenkinsin eduiksi luetaan esimerkiksi

- ilmaisuus
- helppo konfigurointi
- helppo asennus
- jatkuvan kehityksen alainen (kirjoitushetkellä uusin julkaisuversio oli vain 3 päivää vanha)
- suurehko kehittäjäkunta
- Maven-yhteensopivuus

- liitännäisten suuri määrä.

Esimerkiksi Cruise Control -nimiseen vastaavanlaiseen CI-palvelinsovellukseen verrattaessa kohta "helppo konfigurointi" ei täyty, sillä Cruise Controlissa konfigurointi on tehtävä aina kirjoittamalla XML-kuvauskieltä, mutta Jenkinsissä konfigurointi voidaan tehdä graafisen käyttöliittymän kautta. Toinen vahva vaihtoehto on TeamCity, mutta sen ilmaisen Professional-version rajoitukset ovat pitkällä tähtäimellä haitaksi. TeamCityyn ei myöskään ole saatavissa niin runsasta valikoimaa erilaisia liitännäisiä kuin Jenkinsiin. [CI Feature Matrix 2010.] Toisin kuin Jenkins, TeamCity ja Cruise Control eivät kumpikaan ole liitettävissä Trac-nimiseen projektinhallintaohjelmistoon [CI Feature Matrix 2010], joka Playground Finland Oy:llä on käytössään.

5.5 Sonatype m2eclipse

Eclipse on nykyään maailman käytetyin sovelluskehitin Java-kehityksessä [Sonatype 2008: 271] ja sitä käytetään myös Playground Finland Oy:llä sovelluskehitykseen. m2eclipse on Eclipsen liitännäinen, joka helpottaa Mavenin käyttöä graafisen käyttöliittymän avulla. m2eclipsen ominaisuuksia ovat muun muassa Maven-projektien luominen sekä tuominen artefaktivarastosta, riippuvuuksienhallinta, automaattinen riippuvuuksien noutaminen ja päivittäminen, artefaktivarastojen selaus- ja hakutoiminnot sekä riippuvuuksien graafinen esitys. [Sonatype 2008: 271.]

m2eclipseä tarvitaan sovelluskehityksessä esimerkiksi päivitettäessä projektin rakennetta ja riippuvuuksia. m2eclipse ei ole integraatioprosessin kannalta pakollinen työkalu, mutta se helpottaa päivittäistä työtä sekä mahdollistaa uusien asiakaskohtaisten konfiguraatioiden nopean luomisen.

6 Lähtökohdan analysointi

6.1 Lähtökohta

Työn lähtökohtana on Playground Finland Oy:n kehittämä Javalla toteutettu web-sovellus tiedon keräämiseen, hallintaan, esittämiseen sekä jakamiseen. Sillä käsitellään pääasiassa *weblettejä*, jotka ovat yksinkertaisia tiedonsyöttölomakkeesta ja tietokannasta koostuvia sovelluksia. Jokaisella asiakkaalla on käytössään yksi tai useampi weblet.

Sovelluksessa ei ole itsessään tietokantoja tai mahdollisuutta muokata weblettien rakennetta, vaan webletit sijaitsevat *Coressa*, joka on erillinen web-sovellus weblettien rakenteen, käyttöoikeuksien, käyttäjäryhmien ja sisällön muokkaamista varten. Corella on tarkkaan määritelty HTTP-rajapinta, jota käyttämällä voidaan käsitellä weblettien sisältöä ja luoda raportteja weblettien sisällön pohjalta.

Jokaisella asiakkaalla on sovelluksesta oma versionsa eli konfiguraatio, joka on räätälöity kunkin asiakkaan tarpeisiin paremmin soveltuvaksi. Muutokset on kuitenkin toteutettu siten, etteivät ne häiritse sovelluksen itsensä tai muiden asiakkaiden versioiden toimintaa. Asiakaskohtaiset muutokset on tehty joko asiakaskohtaisiin asetustiedostoihin tai asiakaskohtaisiin Java-pakkauksiin. Sovelluksen ydinosa on siis kaikille asiakkaille sama, mutta käytössä oleva asiakaskohtainen konfiguraatio ratkeaa käytössä olevan asetustiedoston perusteella.

Asiakaskohtaisten konfiguraatioiden ainoa pakollinen elementti ydinosan lisäksi on asetustiedosto, mutta ne voivat lähdekoodia sisältävien Java-pakkausten lisäksi sisältää myös lokalisaatiotiedostoja ja teemoja. Lokalisaatiotiedostot sisältävät käännöksiä englannista muille kielille. Kaikki lähdekoodissa olevat käyttäjälle näytettävät tekstit ovat englanniksi, ja mikäli käyttäjän web-selaimen ilmoittamaa kieltä vastaava lokalisaatiotiedosto löytyy, käännetään tekstit ennen niiden näyttämistä sovelluksessa. Teemat voivat sisältää muun muassa asiakasorganisaation logon ja muita sovelluksen ulkoasua muuttavia elementtejä.

Jokaisella asiakkaalla on oma räätälöity sovellusinstanssinsa. Kaikki asiakkaat eivät siis kirjaudu sovellukseen käyttäen samaa URL-osoitetta, vaan jokaisella on oma uniikki URL-osoitteensa, jonka osoittamassa kohteessa sovellus sijaitsee. Kaikki sovelluksen instanssit sijaitsevat samalla Playground Finland Oy:n palvelimella.

Nykymallilla muutosten tekeminen yksittäiseen asiakaskohtaiseen konfiguraatioon tapahtuu seuraavien askelten mukaisesti:

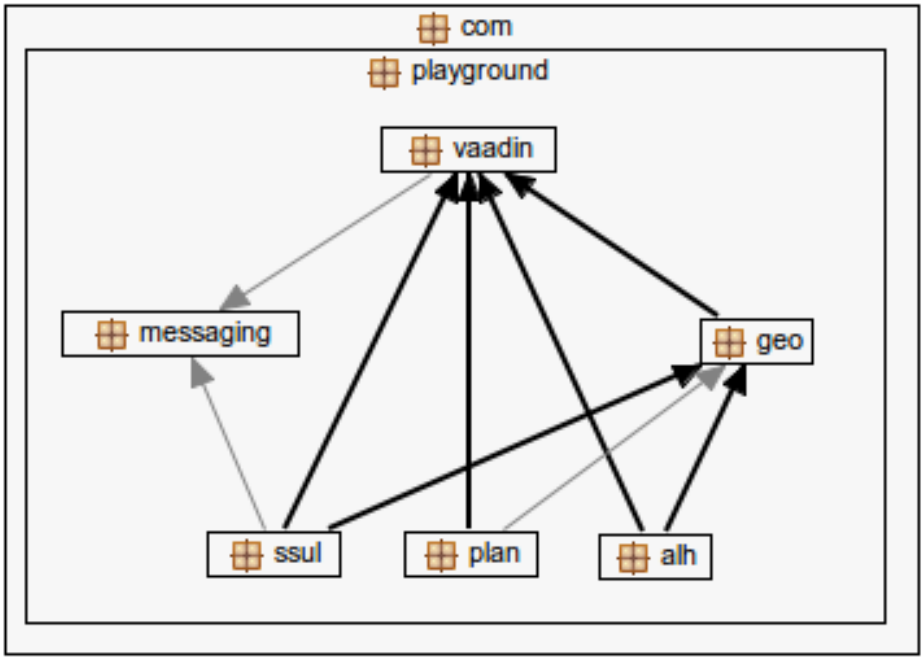
1. Noudetaan sovelluksen uusin versio versionhallinnasta.
2. Kopioidaan asiakaskohtainen asetustiedosto voimassaolevan asetustiedoston päälle.
3. Tehdään muutokset lähdekoodiin.
4. Tallennetaan muutokset versionhallintaan.
5. Paketoidaan sovellus WAR-paketiksi.
6. Lähetetään WAR-paketti sovelluspalvelimelle.
7. Suoritetaan WAR-paketin oikeaan hakemistoon kopioiva käyttöönottoskripti sovelluspalvelimella.

Mikäli kyseessä on muutos, joka on tehtävä kaikkiin asiakaskohtaisiin konfiguraatioihin, on jokainen asiakaskohtainen asetustiedosto otettava yksitellen käyttöön ja suoritettava jokaiselle versiolle askeleet 5–7 uudestaan. Tämä on varsin työlästä, ja tulevaisuudessa asiakaskohtaisten versioiden lisääntyessä työmäärä kasvaisi merkittävästi ilman jatkuvan integraation käyttöönottoa.

6.2 Riippuvuudet

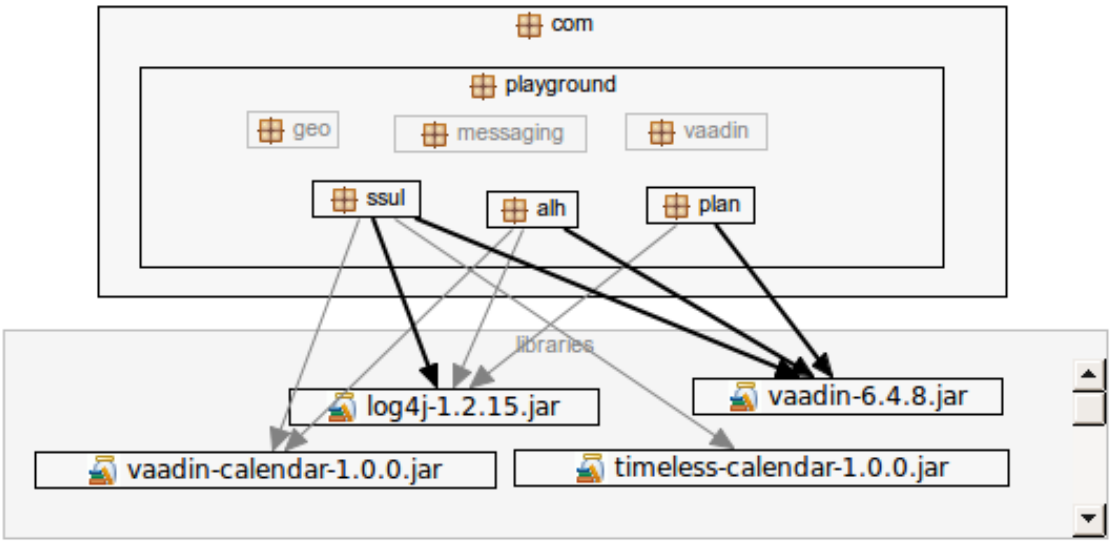
Nykymallilla sovellus on yksi Eclipse-sovelluskehittimen projekti, jossa käytössä oleva asiakaskohtainen konfiguraatio ratkeaa käytössä olevan asiakaskohtaisen asetustiedoston perusteella. Kaikki asiakaskohtaiset elementit ja Java-pakkaukset ovat siis fyysisesti samassa Eclipsen projektissa. Jotta asiakaskohtaiset asetukset ja Java-pakkaukset voitaisiin irrottaa erilleen kyseisestä projektista, on niiden riippuvuudet tunnettava. Riippuvuudet voidaan jakaa kahteen ryhmään: sisäisiin ja ulkoisiin. Sisäisillä riippuvuuksilla tarkoitetaan saman projektin sisäisten moduulien riippuvaisuuksia toisistaan ja ulkoisilla riippuvuuksilla tarkoitetaan riippuvuuksia muun muassa kolmannen osapuolen Java-kirjastoista.

Kuviossa 5 on luvussa 5.1 esiteltyä Ispacea käyttämällä tehty kaavio projektin sisäisistä riippuvuuksista. Kuvion hierarkkisesti kuvatut yksiköt ovat Java-pakkauksen `com.playground` alapakkauksia, joista `ssul`, `plan` ja `alh` ovat asiakaskohtaisia. Pakkaukset `messaging`, `geo` ja `vaadin` muodostavat yhdessä sovelluksen niin sanotun ydinosan. Kuviossa näkyvät nuolet tarkoittavat riippuvuuksia. Esimerkiksi pakkauksen `alh` pakkauksiin `vaadin` ja `geo` osoittamat nuolet tarkoittavat, että pakkaus `alh` on riippuvainen pakkauksista `vaadin` ja `geo`. Harmaa nuoli tarkoittaa yhden Java-luokan riippuvuutta toisen pakkauksen yhdestä Java-luokasta ja musta nuoli useampaa riippuvuutta pakkauksen luokista toisen pakkauksen luokkiin. Kuviosta voidaan havaita, ettei sovelluksessa ole kuvatulla tasolla kaksisuuntaisia riippuvuuksia pakkausten välillä ja täten sitä voidaan kutsua modulaariseksi.



Kuvio 5. Sovelluksen sisäiset riippuvuudet.

Kuviossa 6 on kuvattu asiakaskohtaisten pakkausten riippuvuudet kolmannen osapuolen kirjastoihin. Kuviosta voidaan havaita muun muassa, että kaikki asiakaskohtaiset pakkaukset (`ssul`, `plan`, `alh`) ovat riippuvaisia lokitoimintoja tarjoavan Log4j-kirjaston versiosta 1.2.15 ja Vaadin-sovelluskehyksen versiosta 6.4.8. Kolmannen osapuolen kirjastojen välillä ei ole riippuvuuksia.



Kuvio 6. Sovelluksen ulkoiset riippuvuudet.

Sisäisistä ja ulkoisista riippuvuuksista ja niiden pienestä lukumäärästä voidaan päätellä, ettei riippuvuuksia tarvitse säätää, vaan asiakaskohtaiset komponentit voidaan jaotella suoraan pakkausten mukaisesti. Sovelluksen ydinosan muodostavat pakkaukset `geo`, `messaging` ja `vaadin`.

6.3 Käytettävät palvelimet ja niiden vastuut

Tämän projektin kannalta merkittäviä palvelimia Playground Finland Oy:llä on yhteensä kolme: kehityspalvelin, testipalvelin sekä tuotantopalvelin. Kaikki edellä mainitut ovat skaalautuvia VMware-virtuaalipalvelimia.

Kehityspalvelimella sijaitsee nykyhetkellä Subversion-versionhallintavarasto sekä Trac-projektinhallintaohjelmisto, jota käytetään muun muassa muutostenhallintaan. Palvelimen tarkoituksena on tukea sovelluskehitystä toimimalla kehittäjien välisenä tietovarastona, johon kirjataan korjaus- ja muutospyynnöt, tallennetaan projekteihin liittyvät lähdekoodit, dokumentit ja muut tiedostot sekä pidetään yllä ohjeita ja yleistä tietoa sisältävää wiki-sivustoa.

Testipalvelin on tämän insinöörityön aikana luotu sovellusten testausympäristö, joka on lähestulkoon identtinen tuotantoympäristön kanssa. Sen tarkoituksena on toimia järjestelmätestausympäristönä sekä mahdollistaa uusien toimintojen esittely projektiesimiehille ja tulevaisuudessa mahdollisesti myös asiakkaille.

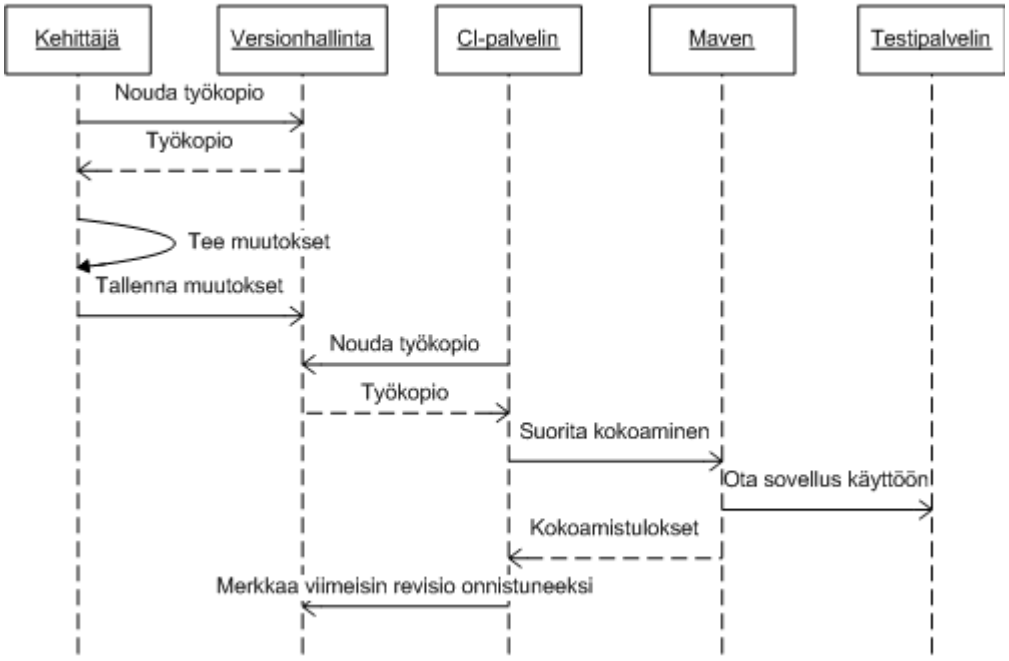
Tuotantopalvelinta käytetään asiakkaiden kaikkien web-sovellusten tuotantoversioiden ajamiseen. Sovelluksia ajetaan käyttämällä Apache Tomcat 6 -sovelluspalvelinohjelmistoa. Tuotantopalvelimeen ei voi luoda suoraa yhteyttä web-selaimella, vaan yhteys luodaan erään toisen Apache HTTP-palvelinohjelmistoa käyttävän palvelimen kautta.

7 Jatkuvan integraatioprosessin luominen

7.1 Jatkuva integraatioprosessi ja sen jälkeiset vaiheet

Jatkuva integraatio on käytäntö ohjelmistokehityksessä ja täten osa ohjelmistokehitysprosessia [Fowler 2006]. Kehitysprosessin lisäksi ohjelmistotuotteen saattaminen kehitysympäristöstä tuotantoon vaatii muitakin vaiheita. Tässä työssä käsiteltävän sovelluksen kohdalla muita vaiheita ovat testaus- ja hyväksyntävaihe sekä toimitusvaihe.

Jatkuvan integraation vaiheet on esitetty yksinkertaistettuna sekvenssikaaviona kuviossa 7. Kaavion kulku esittää häiriötöntä prosessia.



Kuvio 7. Jatkuvan integraation vaiheet kokoamisen onnistuessa

Jatkuvan integraatioprosessin epäonnistuessa Maven ei ota sovellusta testipalvelimelle käyttöön, eikä CI-palvelin (Jenkins) merkkaa koottua revisiota onnistuneeksi. Sen sijaan CI-palvelin lähettää sähköpostia muutoksen tehneelle kehittäjälle ja muille kokoamistyön asetuksiin merkityille vastaanottajille.

Sovelluksen automaattinen käyttöönotto kokoamisen yhteydessä ei välttämättä sovellu kaikenlaisiin projekteihin, sillä meneillään oleva järjestelmä- tai käyttöliittymätestaus keskeytyy, jos toinen kehittäjä tallentaa sovellukseen vaikuttavia muutoksia versionhallintavarastoon ja Maven ottaa sovelluksen uuden version käyttöön automaattisesti. Tässä projektissa riski edellä mainittuun häiriöön on kuitenkin häviävän pieni, sillä kehittäjät työskentelevät harvoin samojen sovelluksen osien parissa, ja täten automaattista käyttöönottoa voidaan hyödyntää.

Sovelluskehitysiteraation päätteeksi sovellukselle suoritetaan järjestelmätestaus. Onnistuneen järjestelmätestauksen jälkeen projektipäällikkö ja muut asianomaiset voivat kokeilla sovelluksen uusia ominaisuuksia testipalvelimella. Inkrementin hyväksymisen jälkeen sovelluksesta tehdään julkaisu- eli tuotantoversio ja otetaan se tuotantopalvelimella käyttöön. Testattua ja hyväksyttyä sovellusversiota ei enää koota uudelleen,

vaan kehitysvaiheessa koottu WAR-paketti otetaan suoraan käyttöön tuotantopalvelimella.

7.2 Artefaktivarastonhallintatyökalun asentaminen ja konfigurointi

Ennen projektin modularisointia ja POM-tiedostojen konfigurointia oli asennettava luvussa 5.3 esitelty artefaktivarastonhallintatyökalu Nexus, sillä sen hallitsemiin artefaktivarastoihin on viitattava POM-tiedostoissa. Nexusia käytetään tässä projektissa julkisten artefaktivarastojen välityspalvelimena sekä yrityksen sisäisten artefaktivarastojen hallintaan.

Kuten luvussa 5.3 mainittiin, organisaation sisäinen artefaktivarastojen välityspalvelin nopeuttaa artefaktien lataamista. Tästä saavutettava hyöty jää kuitenkin Playground Finland Oy:llä vähäiseksi, sillä yritys on virtuaaliorganisaatio, eikä artefaktivaraston välityspalvelin ole täten yhdenkään kehittäjän kanssa samassa lähiverkossa. Siitä huolimatta Nexusia tarvitaan organisaation sisäisten artefaktien jakamisen lisäksi sellaisten sovelluksissa tarvittavien kolmannen osapuolen kirjastojen jakamiseen, jotka eivät ole saatavilla julkisissa artefaktivarastoissa.

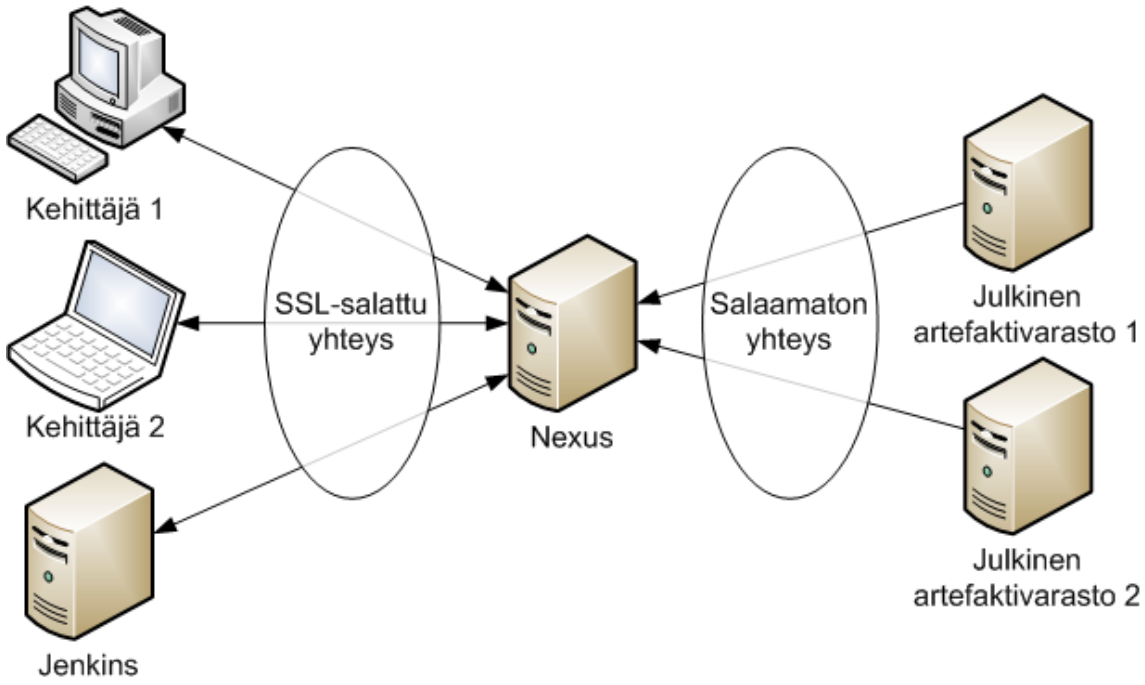
Ennen Nexusin asentamista kehityspalvelimelle asennettiin Apache Tomcat 6 -sovelluspalvelinohjelmisto, sillä Nexusin lisäksi myös Jenkinsiä voidaan ajaa Tomcatissa. Tomcat ei ole Nexusin ja Jenkinsin ajamiseen pakollinen, sillä molempia voidaan myös käyttää niiden mukana toimitettavissa sovelluspalvelinohjelmistoissa. Useamman Javalla toteutetun web-sovelluksen ajaminen yhden sovelluspalvelinohjelmiston sisällä on suotavaa, sillä se helpottaa sovellusten hallintaa. Tomcatia käytettäessä Nexusin asentaminen ja käyttökuntoon saattaminen ei vaatinut muuta kuin asennuspaketin kopioimisen Tomcatin sovellushakemistoon.

Yrityksen sisäisiksi artefaktivarastoiksi määritettiin artefaktivarastot Releases, Snapshots ja 3rd party. Releases-artefaktivarastoon on tarkoitus laittaa artefaktien vakaaksi katsotut julkaisuversiot, joihin voidaan viitata muiden artefaktien kehitys- ja julkaisuversioista. Snapshots sisältää artefaktien kehitysversiot, joita muut kehittäjät saattavat tarvita muissa samaa artefaktia käyttävissä projekteissa. 3rd party on tarkoi-

tettu kolmannen osapuolen kirjastoille, joita ei löydy julkisista artefaktivarastoista tai joista on julkisissa artefaktivarastoissa saatavilla vain vanhoja versioita.

Käytännössä jokaisella tietokoneella, johon Maven on asennettu, on paikallinen artefaktivarasto. Kertaalleen ladattuja artefakteja ei siis tarvitse ladata uudestaan etäartefaktivarastoista. [Introduction to Repositories 2011.]

Kuviossa 8 on esitetty kehittäjien, CI-palvelimen (Jenkins), yrityksen artefaktivarastojen (Nexus) ja julkisten artefaktivarastojen yhteys toisiinsa. Kehittäjien ja yrityksen artefaktivarastojen välinen yhteys on salattu, koska sen kautta siirretään yrityksen yksityisiä artefakteja.



Kuvio 8. Artefaktivarastonhallintatyökalu välityspalvelimena sekä sisäisenä artefaktivarastona.

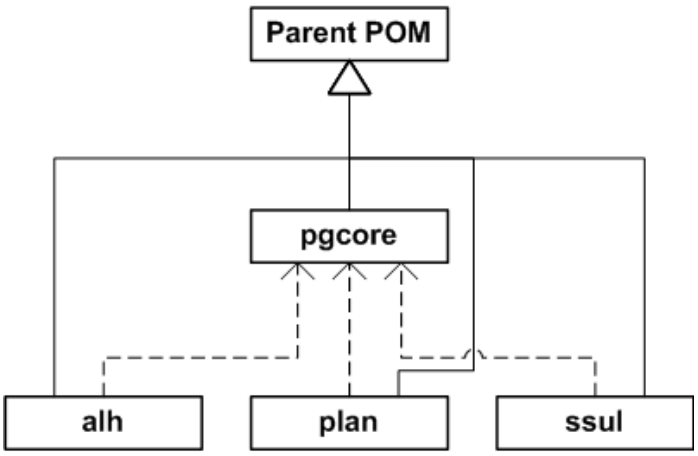
Kehittäjien ja sisäisten artefaktivarastojen välisen yhteyden salaamisen lisäksi artefaktinhallintatyökalu vaatii autentikaation kaikkien artefaktivarastojen osalta. Täten myös julkisista artefaktivarastoista ladattujen artefaktien lataaminen vaatii autentikaation. Autentikaatiolla vältetään artefaktivarastonhallintatyökalun väärinkäyttö. Käyttäjätunnukset ja -oikeudet määritettiin Nexusin asetuksiin siten, että kaikilla käyttäjillä on luku- ja kirjoitusoikeudet sisäisiin artefaktivarastoihin.

7.3 Konfiguraatioiden määrittäminen

Luvussa 6.2 luotua sovelluksen riippuvuuskaaviota apuna käyttäen voitiin asiakaskohtaiset komponentit tunnistaa ja irrottaa sovelluksen ydinosasta. Riippuvuuskaavion fokus on asiakaskohtaisissa lähdekooditiedostoja sisältävissä Java-pakkauksissa, mutta koska kaikkien asiakkaiden sovellusversioissa ei ole sovelluksen ydinosasta poikkeavaa lähdekoodia, ei niitä myöskään näy riippuvuuskaaviossa. Siitä huolimatta kaikilla asiakkailla on oltava oma Maven-artefaktinsa eli konfiguraatio.

Asiakaskohtaisen konfiguraation sisällön vähimmäisvaatimus on ydinosan lisäksi sovelluksen asetustiedosto. Ilman asiakaskohtaista asetustiedostoa sovellusta voidaan käyttää oletusasetuksin ilman mitään asiakasspesifejä modifikaatioita, kuten ulkoasua, logoa tai muutettua toiminnallisuutta. Ydinosa on siis itsenäisesti toimiva ja asiakaskohtaisista komponenteista riippumaton sovellus, joka sisältää lähdekoodia, lokiasetustiedoston, mukautetun ulkoasun, suurehkon valikoiman ikoneita sekä sovelluksen asetustiedoston.

Jokainen konfiguraatio vaatii Mavenia käytettäessä oman POM-tiedostonsa. Saman tiedon toistamisen ja virhealttiuden välttämiseksi jokaisen konfiguraation POM periytyy yhteisestä Parent POM-tiedostosta, joka on listattuna liitteessä 1. Parent POM sisältää tietoa muun muassa artefaktivarastoista, liitännäisistä sekä riippuvaisuuksista, jotka ovat kaikille konfiguraatioille yhteisiä. Kuviossa 9 on kuvattu konfiguraatioiden riippuvuudet katkoviivalla ja periytyminen yhtenäisellä viivalla.



Kuvio 9. Konfiguraatioiden riippuvuudet ja periytyminen

Kaikki asiakaskohtaiset konfiguraatiot ovat suoraan riippuvaisia sovelluksen ydinosasta eli `pgcore`-konfiguraatiosta. Jotta asiakaskohtaisten konfiguraatioiden käännökset onnistuisivat ja kokoamisprosessin lopputulos toimisi oikein, on `pgcore`-konfiguraatiosta sen kokoamisvaiheessa luotava paketit JAR- sekä WAR-muodoissa. JAR-paketti sisältää ainoastaan `pgcore`-konfiguraation lähdekoodin, ja WAR-paketti sisältää lähdekoodin lisäksi muita sovelluksen komponentteja, kuten oletusasetustiedoston, ikoneita ja mukautetun ulkoasun. `pgcore`-konfiguraation JAR-pakettia tarvitaan asiakaskohtaisen konfiguraation kokoamisprosessin käännösvaiheessa vain silloin, kun asiakkaan konfiguraatio sisältää sellaisia lähdekoodikomponentteja, jotka ovat riippuvaisia `pgcore`-konfiguraation sisältämästä lähdekoodista. WAR-pakettia ei hyödynnetä käännösvaiheessa lainkaan, sillä sen sisältämä hakemistorakenne poikkeaa JAR-paketista, minkä takia Javan kääntäjä ei osaa käsitellä WAR-pakettia riippuvuutena JAR-paketin tavoin. `pgcore`-konfiguraation kokoamisprosessin tuottaman WAR-paketin sisältö kuitenkin yhdistetään asiakaskohtaisen konfiguraation WAR-pakettiin käyttäen luvussa 5.2.2 esiteltyä overlay-mekanismia. `pgcore`-konfiguraation POM-tiedostossa projektin tyypiksi on määritelty WAR, eli kokoamisprosessi tuottaa lopulta WAR-muotoisen paketin. JAR-paketin luominen edellyttää Mavenin WAR-liitännäisen konfigurointia, joka on kuvattuna listauksessa 1.

```
<plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.1.1</version>
    <configuration>
        <attachClasses>true</attachClasses>
    </configuration>
</plugin>
```

Listaus 1. JAR-paketin muodostamiseen tarvittava konfiguraatio

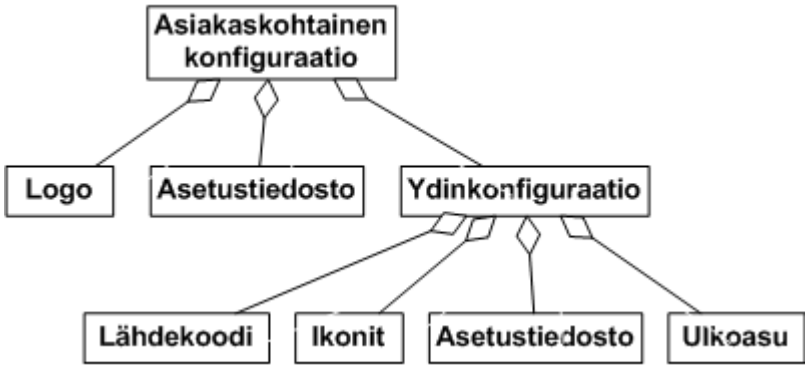
Riippuvuus `pgcore`-konfiguraatioon on määritelty kahdella tavalla asiakaskohtaisten konfiguraatioiden POM-tiedostoissa. Riippuvuuksien määritteleminen on kuvattu listauksessa 2.

```
<dependency>
    <groupId>com.playground</groupId>
    <artifactId>pgcore</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <type>jar</type>
    <classifier>classes</classifier>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>com.playground</groupId>
    <artifactId>pgcore</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <type>war</type>
    <scope>runtime</scope>
</dependency>
```

Listaus 2. Riippuvuudet pgcoreen asiakaskohtaisen konfiguraation POM-tiedostossa

Listauksessa 2 ensin määritellyssä riippuvuudessa on määritelty riippuvuuden tyypiksi `jar`, luokittelijaksi `classes` ja näkyvyysalueeksi `provided`. Luokittelija tarkoittaa JAR-tiedoston perään lisättyä jälkiliitettä, joka on WAR-liitännäisessä oletuksena arvoltaan `classes`. Tällöin `pgcore`-konfiguraatiosta luodun JAR-paketin nimi on muodossa `pgcore-1.0.0-SNAPSHOT-classes.jar`. Näkyvyysalue `provided` tarkoittaa sitä, että riippuvuus on mukana käännösvaiheessa, mutta sitä ei enää paketoida lopulliseen artefaktista muodostettavaan pakettiin [Sonatype 2008: 160]. Riippuvuuden JAR-paketti jätetään muodostettavasta asiakaskohtaisen konfiguraation WAR-paketista tässä tapauksessa pois, koska JAR-paketin sisältämät Java-luokat ovat joka tapauksessa jo kuvion 10 jälkimmäiseksi määritellyssä WAR-riippuvuudessa mukana. WAR-riippuvuuden näkyvyysalue `runtime` tarkoittaa, että WAR-paketin tiedostoja tarvitaan vain testaamiseen ja ajonaikaisesti [Sonatype 2008: 160]. Tällöin niitä ei hyödynnetä käännösvaiheessa, mutta paketointivaiheessa suoritetaan WAR-liitännäisen overlay-toiminto, joka yhdistää sovellusten komponentit samaan WAR-pakettiin.

Asiakaskohtainen konfiguraatio koostuu ydinosasta, joka on komponentteja sisältävä konfiguraatio, sekä yhdestä tai useammasta asiakaskohtaisesta komponentista. Konfiguraation rakenne on määritelty POM:ssa. Komponenttien sijaintia ei tarvitse määritellä POM:ssa erikseen, sillä ne sijaitsevat Mavenin konvention määrittelemissä hakemistoissa. Esimerkki asiakaskohtaisen konfiguraation rakenteesta on havainnollistettuna kuviossa 10. Esimerkissä ydinkonfiguraation asetustiedosto jää hyödyntämättä WAR-liitännäisen overlay-toiminnon takia.



Kuvio 10. Esimerkki asiakaskohtaisen konfiguraation rakenteesta

Yhteys yrityksen sisäisiin ja artefaktivarastonhallintatyökalun välittämiin artefaktivarastoihin vaatii SSL-salauksen sekä palvelinsertifikaattiin luottamisen. Java-sovellusten luodessa SSL-salatun yhteyden, on kohdepalvelimen sertifikaatti oltava sovellusta ajavan ja yhteyden ottavan tietokoneen Java-avainsäilössä. Avainsäilössä olevaa sertifikaattia kutsutaan luotetuksi sertifikaatiksi.

Kehityspalvelimen sertifikaatti lisättiin avainsäilöön käyttäen InstallCert-nimistä Java-sovellusta. InstallCert kopioi aiemmin luodusta avainsäilöstä kaikki luotetut sertifikaatit uuteen avainsäilöön ja lisää sitten sille parametrina annetusta URL-osoitteesta löytyvän SSL-sertifikaatin uuteen avainsäilöön. InstallCertiä käytetään komentoriviltä kutsuen sen komentoa muodossa `java InstallCert palvelin`, jossa parametri `palvelin` on SSL-sertifikaatin tarjoavan palvelimen URL-osoite. Lopuksi InstallCertin luoma uusi avainsäilö `jssecacerts` kopioidaan Javan ajoympäristön `lib/security` -hakemistoon. [Sterbenz 2006.]

Yhteyden muodostaminen artefaktivarastoihin vaatii SSL-salauksen lisäksi myös autentikaation. Autentikaatio ja käyttäjätasot konfiguroitiin Nexusiin siten, ettei yhteyttä voi muodostaa ilman artefaktivaraston lukuoikeuksia eikä ilman asianmukaista käyttäjätunnusta ja salasanaa. Käyttäjätunnus on määritettävä yhteyden ottavan tietokoneen `settings.xml`-nimiseen Mavenin asetustiedostoon. Asetustiedosto voidaan konfiguroida järjestelmä- tai käyttäjäkohtaisesti. Linux-jakeluissa järjestelmäkohtainen asetustiedosto sijaitsee Mavenin asennushakemiston `conf`-alihakemistossa ja mahdollinen käyttäjäkohtainen asetustiedosto on sijoitettava käyttäjän kotihakemiston `.m2`-alihakemistoon. Käyttäjätunnus ja salasana konfiguroidaan palvelinkohtaisesti asetustiedostoon listauksen 3 mallin mukaan.

```
<server>
    <id>pg-snapshots</id>
    <username>käyttäjätunnus</username>
    <password>salasana</password>
</server>
```

Listaus 3. Käyttäjätunnus ja salasana Mavenin asetustiedostossa

Asetustiedoston palvelinkonfiguraation `id`-elementtiin merkittiin artefaktivaraston tunnus, joka on identtinen Parent POM:ssa määritellyn artefaktivaraston tunnuksen kanssa. Artefaktivaraston URL-osoite on määritelty Parent POM:ssa. Vaikka artefaktivarastoja on useampia, riitti yhden artefaktivaraston käyttäjätunnuksen ja salasanan määrittely asetustiedostoon, sillä kaikki muut Parent POM:ssa määritellyt artefaktivarastot sijaitsevat loogisesti samalla palvelimella.

Lopuksi asiakaskohtaisille konfiguraatioille sekä ydinosalle luotiin modularisointivaiheessa omat hakemistonsa versionhallintavarastoon niin sanotun trunk-hakemiston alle. Trunkilla tarkoitetaan versionhallinnan päähaaraa.

7.4 CI-palvelimen asentaminen ja konfigurointi

Kuten Nexusinkin kohdalla, Jenkinsin asennus kehityspalvelimelle ei vaatinut muuta kuin asennuspaketin kopioimisen Tomcat-sovelluspalvelinohjelmiston sovellushakemistoon. Jenkinsin asennus ja avaaminen web-selaimessa kesti alle viisi minuuttia.

Jenkins konfiguroitiin käyttämään autentikointiin pwauth-sovellusta, joka sallii kehityspalvelimen Unix-käyttäjätunnusten käyttämisen kirjautumisessa. Käyttöoikeustasot jaettiin Unix-käyttäjäryhmien perusteella kolmeen osaan; järjestelmänvalvojaryhmään kuuluvilla on oikeudet suorittaa kaikki mahdolliset toimenpiteet, kehittäjät voivat käynnistää kokoamisen, luoda omia kokoamistyönäkymiä sekä merkitä kokoamistöitä versionhallintavarastoon, ja anonyymeilla eli kirjautumattomilla käyttäjillä ei ole Jenkinsin käyttöoikeuksia laisinkaan.

Jenkinsin suorittamat kokoamisprosessit vaativat SSL-sertifikaatin ja käyttäjätunnukset artefaktivarastoihin. Molemmat lisättiin CI-palvelimelle edellisessä luvussa esitettyjen ohjeiden mukaisesti. Avainsäilö tallennettiin CI-palvelimelle oletusarvosta poikkeavaan

sijaintiin, joka määritettiin Jenkinsin asetuksiin. Avainsäilön sijainnin määrittämisen johdosta mahdolliset Java-ajoympäristön versiopäivitykset eivät vaadi avainsäilön kopiointia uuden ajoympäristön hakemistoon.

Jenkinsiin asennettiin joukko liitännäisiä kokoamisprosessia ja Jenkinsin käyttöä varten. Edellä mainittu pwauth-autentikaatio vaati pwauth-liitännäisen, joka lisäsi Jenkinsin asetuksiin valinnan tätä autentikaatiotapaa varten. Jenkinsin käyttöä varten asennettiin pwauth-liitännäisen lisäksi Trac-liitännäinen, joka luo Jenkinsin kokoamistöihin linkkejä kehityspalvelimella sijaitsevaan Trac-projektinhallintaohjelmistoon ja sen tarjoamiin resursseihin.

Kokoamisprosessia varten asennettiin

- Subversion-liitännäinen, jonka avulla kokoamista varten tarvittavat tiedostot voidaan noutaa Subversion-versionhallintavarastosta.
- Subversion tagging -liitännäinen, joka merkitsee viimeisimmän onnistuneen kokoamisen kokoamistyökohtaisesti versionhallintavaraston tags-haaraan.
- FindBugs-liitännäinen, joka tekee Java-lähdekoodille staattisen analyysin ja raportoi käännöksen lopuksi lähdekoodin mahdollisista virheistä.
- Copy Artifact -liitännäinen, jonka tehtävänä on kopioida yhden kokoamistyön artefakteja toiseen kokoamistyöhön.

Liitännäisten asentaminen on yksinkertaista; asennettavat liitännäiset valitaan listasta ja lopuksi painetaan asennuspainiketta. Mikäli jotain tiettyä liitännäistä ei löydy listasta, sen voi asentaa lataamalla asennuspaketin liitännäisen web-sivustolta omalle tietokoneelle ja asentaa sen Jenkinsin liitännäisten asennussivun lomaketta käyttäen.

Eri konfiguraatioiden kokoamiset suoritetaan Jenkinsissä niille omistetuissa kokoamistöissä. Kokoamistyön voi itse kokoamisen lisäksi konfiguroida suorittamaan muitakin Jenkinsissä monitoroitavia tehtäviä, kuten erilaisia testejä, tiedostojen kopioimista sekä ohjelmistotuotteiden käyttöönottoa. Kokoamistyö voidaan käynnistää esimerkiksi Jenkinsin web-käyttöliittymästä manuaalisesti tai työlle määritettyjen herätteiden aktivoituessa. Heräte voi aktivoitua esimerkiksi työlle asetetun ajastuksen määrittämänä aikana tai työn sisältämän Maven-artefaktin kehitysversioriippuvuuden päivittyessä. Ko-

koamistyö voi myös seurata versionhallintavaraston tilaa aika-ajoin ja suorittaa tehtävänsä, kun seurattavat tiedot versionhallintavarastossa päivittyvät.

Kokoamistyöt näytetään Jenkinsin web-käyttöliittymässä listoissa, joita kutsutaan näkymiksi. Näkymä on ikään kuin kategoria, johon kokoamistyö kuuluu. Näkymiä luotiin tätä projektia varten kolme; pelkkä kokoaminen, kokoaminen ja käyttöönotto testiympäristöön, ja käyttöönotto tuotantoympäristöön. Jenkinsin oletusnäkymä listaa kaikki kokoamistyöt. Kuviossa 11 on ruudunkaappaus Jenkinsiin luodusta näkymästä.

Kuvio 11. Jenkinsin käyttöliittymä

Kuviossa 11 esitettävän näkymän ensimmäinen sarake kertoo viimeisimmän kokoamisen tilan. Kokoamisen epäonnistuessa sarakkeessa näkyvä ympyrä muuttuu punaiseksi ja onnistuessa siniseksi. Toisessa sarakkeessa kuvataan viiden viimeisen työsuorituksen tila sääsymboleilla; aurinko tarkoittaa viiden suorituksen onnistumista ja ukkospilvi viiden suorituksen epäonnistumista. Näiden tilojen välillä on myös pouta- ja sadepilviä, jotka myös helpottavat työsuoritusten tilan hahmottamista kokonaisuutena.

Kokoamistöiden asetukset ovat keskenään näkymäkohtaisesti hyvin samankaltaisia. Ohessa esimerkki erään koottavan ja testiympäristöön käyttöönotettavan kokoamistyön asetuksista:

- Poistetaan yli viikon vanhat kokoamistyösuoritteet. Tuotantoon otetut kokoamistyösuoritteet säilyvät Jenkinsissä tästä huolimatta.
- Tarkkaillaan versionhallintavarastossa konfiguraatiokohtaiseen hakemistoon tehtäviä muutoksia viiden minuutin välein ja noudetaan mahdolliset muutokset

päivittämällä työkopio. Käynnistetään kokoamistyö työkopion päivittämisen jälkeen.
- Käynnistetään kokoamistyö, jos kokoamistyön sisältämän Maven-artefaktin kehitysversioriippuvuus kootaan samassa Jenkins-instanssissa.
- Julkaistaan FindBugs-liitännäisen tuottamat analyysitulokset.
- Ilmoitetaan viimeisimmän muutoksen tehneelle ja muille luetelluille asianomaisille työn epäonnistumisesta ja epäonnistumista seuraavasta onnistumisesta sähköpostitse.
- Merkitään onnistunut kokoaminen luomalla versionhallintavarastoon merkintä hakemistoon `/tags/last-successful/<kokoamistyön_nimi>`.

Edellisten lisäksi on Mavenin komennoksi määritelty seuraavat parametrit, maalit sekä vaiheet alla esitetyssä järjestyksessä:

1. `-Ddeploy=uat` on parametri, joka aktivoi testiympäristön Maven-profiilin, joka kertoo sovelluksen käyttöönoton hoitavalle Cargo-liitännäiselle käyttöönottokohteen URL-osoitteen. Tuotantoympäristön profiili aktivoituu korvaamalla parametrin arvo `uat` arvolla `production`.
2. `clean` on vaihe, joka tyhjentää kokoamisen kohdehakemiston.
3. `vaadin:update-widgetset` on maali, joka päivittää Vaadin-sovelluskehyksen selainpuolelle kohdistettua lähdekoodia sisältävien lisäosien listan artefaktin riippuvuuksien perusteella.
4. `gwt:compile` on maali, joka käynnistää Google Web Toolkitin kääntäjän, joka muun muassa kääntää edellä mainittujen lisäosien sisältämää Java-lähdekoodia web-selaimille tarkoitetuksi JavaScript-lähdekoodiksi.
5. `install` on vaihe, johon päästessä artefaktin lähdekoodi on käännetty, testattu ja asennettu paikalliseen Maven-artefaktivarastoon.
6. `findbugs:findbugs` on maali, joka suorittaa artefaktin lähdekoodille staattisen analyysin, eli etsii koodista mahdollisia virheitä.
7. `cargo:redeploy` on maali, joka suorittaa artefaktin käyttöönoton `deploy`-parametrin määrittämään ympäristöön.

Edellä esitetyn kaltaisen kokoamistyön suorittaminen kestää keskimäärin noin 45 sekuntia. Työ voi tosin kestää jopa kuusi minuuttia silloin, kun edellä mainittu

`gwt:compile` -maali tunnistaa uutta JavaScript-käännöksen tarvitsevaa lisäosan Java-lähdekoodia.

Kaikkiin kokoamisen jälkeen testiympäristössä käyttöönotettaviin kokoamistöihin määriteltiin heräte, joka aktivoituu artefaktin kehitysversioriippuvuuden muuttuessa. Koska jokainen asiakaskohtainen konfiguraatio on tällä hetkellä riippuvainen sovelluksen ydinosan kehitysversiosta, käynnistyy jokaisen asiakaskohtaisen konfiguraation kokoamistyö heti ydinosan onnistuneen kokoamistyön valmistuttua. Täten ydinosaan tehdyt muutokset ovat erittäin nopeasti kokeiltavissa testiympäristössä kaikkien eri asiakkaiden sovelluksissa.

Sovellusten käyttöönotto tuotantoympäristöön konfiguroitiin tapahtumaan Jenkinsin Copy Artifact -liitännäisen ja Mavenin Cargo-liitännäisen avulla. Käyttöönotettavan sovelluksen kokoamisen ja käyttöönoton testiympäristöön sisältävä kokoamistyö merkitään merkinnällä "säilytä ikuisesti", jonka jälkeen sovelluksen tuotantoympäristöön käyttöönottava kokoamistyö voidaan käynnistää. Kyseinen kokoamistyö poimii siis viimeisimmän "säilytä ikuisesti" -merkityn kokoamistyön, kopioi sen POM-tiedoston ja WAR-paketin omaan työtilaansa ja suorittaa käyttöönoton Mavenin Cargo-liitännäisellä. Mavenin komento koostuu tässä kokoamistyössä parametrista `-Ddeploy=production`, joka aktivoi Maven-profiilin, joka puolestaan kertoo Cargo-liitännäiselle käyttöönoton kohdeympäristön URL-osoitteen, ja maalista `cargo:redeploy`, joka pyytää Cargo-liitännäistä suorittamaan käyttöönoton.

Tarpeen vaatiessa on mahdollista ottaa myös jonkin muun kokoamistyösuorituksen tulos käyttöön tuotantoympäristössä muuttamalla tuotantoympäristöön käyttöönottavan kokoamistyön asetuksia väliaikaisesti ja määrittämällä kokoamistyön tunnuksen. Näin voidaan menetellä esimerkiksi silloin, kun tuotantoympäristöstä löytyy virhe, jonka johdosta edellinen tuotantoversio on otettava pikaisesti takaisin käyttöön.

Kopiointi- ja käyttöönottotoimenpiteiden lisäksi kokoamistyö tallentaa käyttöönotettavan WAR-paketin "sormenjäljet" paketin myöhempää tunnistamista varten. Tuotantokäytössä oleva WAR-paketti voidaan tunnistaa sormenjälkien avulla, josta voi olla hyötyä esimerkiksi virheiden jäljittämisessä tai tietyn tuotannossa olleen version riippuvuuksien selvittämisessä.

8 Yhteenveto

Tässä insinöörityössä ratkaistiin sovelluksen kasvaneesta kompleksisuudesta johtunut manuaalisen työn lisääntyminen jakamalla lähdekoodin osat tarkemmin määritellyiksi ohjelmakokonaisuuksiksi, määrittelemällä komponenttien riippuvuudet toisistaan ja kolmansien osapuolien kirjastoista sekä automatisoimalla käyttöönottovaiheessa tapahtuva käännös, integraatio ja käyttöönotto. Työ onnistui odotetulla tavalla.

Kehitys- ja toimitusprosessi olivat ennen insinöörityön toteuttamista hieman vaatimattomampia; tehdyt muutokset tallennettiin versionhallintavarastoon ja sovelluksen uusi asiakaskohtainen versio otettiin käyttöön tuotantopalvelimella. Tämä menetelmä kävi kuitenkin sovelluksen asiakaskohtaisten muunnosten lisääntyessä työlääksi, ja lisääntynyt käsin tehtävä työ altisti sovelluksen ja sen toimitusprosessin virheille.

Insinöörityön toteuttamisen jälkeen sovelluksen saattamisesta kehitysympäristöstä tuotantoon tuli järjestelmällisempää, vähemmän manuaalista työtä vaativaa sekä vähemmän virhealtista. CI-palvelimen ansiosta sovelluksen virheet voidaan löytää aikaisemmassa vaiheessa ja asiakaskohtaisten konfiguraatioiden terveydentilaa, eli toimivuutta ja laatukriteereiden täyttymistä, voidaan seurata lähestulkoon reaaliajassa. Myös sisäisten ja ulkoisten riippuvuuksien hallinta sekä ristiriitojen selvittäminen helpottui merkittävästi.

Insinöörityön tuloksena syntynyttä sovelluksen uutta rakennetta ja jatkuvaa integraatioprosessia sekä niihin tarvittavia työkaluja on käytetty vasta vähän aikaa, joten on mahdollista, että rakennetta ja prosessia tullaan vielä hienosäätämään. Esimerkiksi CI-palvelimen suorittaman herätteellisen kokoamisen jälkeen tapahtuva automaattinen käyttöönotto testiympäristöön saatetaan ottaa pois käytöstä, mikäli kehittäjien ja versionhallintavarastoon tehtävien muutosten määrä kasvaa tulevaisuudessa niin paljon, että se häiritsisi järjestelmätestaamista.

Lähteet

Aracic, Ivica. 2008. Verkkodokumentti. Ispace user guide. <http://ispace.stribor.de/index.php?n=Ispace.UserGuide>. Luettu 14.1.2011.

Aracic, Ivica. 2010. Verkkodokumentti. About Ispace. <http://ispace.stribor.de/index.php?n=Ispace.Home>. 21.10.2010. Luettu 14.1.2011.

Bayer, Andrew. 2011a. Verkkodokumentti. Hudson's future. <http://www.hudson-labs.org/content/hudsons-future>. 11.1.2011. Luettu 15.1.2011.

Bayer, Andrew. 2011b. Verkkodokumentti. Jenkins!. <http://jenkins-ci.org/content/jenkins>. 29.1.2011. Luettu 5.2.2011.

CI Feature Matrix. 2010. Verkkodokumentti. <http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix>. 29.12.2010. Luettu 15.1.2011.

Duvall, Paul M. & Matyas, Steve & Glover, Andrew. 2009. Continuous Integration. Improving software quality and reducing risk. Boston: Pearson Education.

Fowler, Martin. 2006. Verkkodokumentti. Continuous Integration. <http://www.martinfowler.com/articles/continuousIntegration.html>. 1.5.2006. Luettu 10.1.2010.

Haikala, Ilkka & Märijärvi, Jukka. 2006. Ohjelmistotuotanto. 11. painos. Helsinki: Talentum.

Introduction to the Build Lifecycle. 2011. Verkkodokumentti. Apache Maven Project. <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>. 12.1.2011. Luettu 14.1.2011.

Introduction to Repositories. 2011. Verkkodokumentti. <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>. 31.1.2011. Luettu 31.1.2011.

Iteratiivinen ja inkrementaalinen kehitys. 2008. Verkkodokumentti. Ketterät käytännöt. <http://www.ketteratkaytannot.fi/Ketteryys/IteraatiotJaInkrementit/>. Luettu 17.12.2010.

Ketterät käytännöt. 2008. Verkkodokumentti. Ketterät käytännöt. <http://www.ketteratkaytannot.fi/fi-FI/>. Luettu 17.12.2010.

Koskela, Juha. 2003. Software configuration management in agile methods. Espoo: VTT Publications.

Kosonen, Pekka & Peltomäki, Juha & Silander, Simo. 2007. Java 2. Ohjelmoinnin peruskirja. 4. laitos. Jyväskylä: WSOYpro.

Maven Repository Manager Feature Matrix. 2011. Verkkodokumentti. <http://docs.codehaus.org/display/MAVENUSER/Maven+Repository+Manager+Feature+Matrix>. 7.1.2011. Luettu 29.1.2011.

Maven WAR Plugin. 2010. Verkkodokumentti. <http://maven.apache.org/plugins/maven-war-plugin>. 3.11.2010. Luettu 4.2.2011.

Meet Jenkins. 2011. Verkkodokumentti. Jenkins CI. <http://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>. 2.2.2011. Luettu 5.2.2011.

O'Sullivan, Bryan. 2009. Mercurial: The Definitive Guide. Sebastopol: O'Reilly Media.

Overlays. 2010. Verkkodokumentti. <http://maven.apache.org/plugins/maven-war-plugin/overlays.html>. 3.11.2010. Luettu 4.2.2011.

Perry, Mike & Oskov, Nasko. 2004. Verkkodokumentti. Introduction to Reverse Engineering Software. <http://www.acm.uiuc.edu/sigmil/RevEng/ch02.html>. 12.2.2004. Luettu 9.1.2011.

Pienet julkaisut. 2008. Verkkodokumentti. Ketterät käytännöt. <http://www.ketteratkaytannot.fi/fi-FI/Kaytannot/PienetJulkaisut/>. Luettu 18.12.2010.

Sommerville, Ian. 2007. Software Engineering. 8th edition. Essex: Pearson Education.

Sonatype. 2008. Maven: The Definitive Guide. First Edition. Sebastopol: O'Reilly Media.

Sterbenz, Andreas. 2006. Verkkodokumentti. <http://blogs.sun.com/andreas/entry/no_more_unable_to_find>. 9.10.2006. Luettu 5.2.2011.

Using Vaadin with Maven. 2010. Verkkodokumentti. Vaadin. <http://vaadin.com/wiki/-/wiki/Main/Using%20Vaadin%20with%20Maven>. 21.9.2010. Luettu 15.1.2011.

Vuori, Päivi. 2009. Development of agile software production to the organisation. Opinnäytetyö. Tampereen ammattikorkeakoulu.

What Is Maven?. 2011. Verkkodokumentti. Apache Maven Project. <http://maven.apache.org/what-is-maven.html>. 12.1.2011. Luettu 14.1.2011.

Wiest, Simon. 2010. Verkkodokumentti. Hudson - Your Escape from "Integration Hell". <http://www.methodsandtools.com/tools/tools.php?hudson>. Luettu 15.1.2011.

Parent POM -tiedoston sisältö

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>com.playground</groupId>
 <artifactId>parentpom</artifactId>
 <version>1.0.0</version>
 <packaging>pom</packaging>
 <name>PG Parent POM</name>

 <description>The parent POM for all PG artifacts</description>

 <ciManagement>
  <system>Jenkins</system>
  <url>https://devserver/ci</url>
 </ciManagement>

 <organization>
  <name>Playground Finland Oy</name>
  <url>http://www.anyplayground.com</url>
 </organization>

 <inceptionYear>2010</inceptionYear>

 <issueManagement>
  <system>trac</system>
  <url>https://devserver/projects/geo</url>
 </issueManagement>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

<profiles>
 <profile>
  <id>testdeploy</id>
  <activation>
   <property>
    <name>deploy</name>
    <value>uat</value>
   </property>
  </activation>
  <properties>
   <tomcat.url>http://uatserver:8080/manager</tomcat.url>
  </properties>
 </profile>
 <profile>
  <id>productiondeploy</id>
  <activation>
   <property>
    <name>deploy</name>
    <value>production</value>
   </property>
  </activation>
```

```
  <properties>
   <tomcat.url>http://productionserver:8080/manager</tomcat.url>
  </properties>
 </profile>
</profiles>

<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>2.3.2</version>
   <configuration>
    <source>1.6</source>
    <target>1.6</target>
   </configuration>
  </plugin>

  <plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>gwt-maven-plugin</artifactId>
   <version>2.1.0-1</version>
   <configuration>
    <webappDirectory>
     ${project.build.directory}/${project.build.finalName}/VAADIN/
     widgetsets
    </webappDirectory>
    <extraJvmArgs>-Xmx512M -Xss1024k</extraJvmArgs>
    <runTarget>${gwt.maven.runtarget}</runTarget>
    <hostedWebapp>
     ${project.build.directory}/${project.build.finalName}
    </hostedWebapp>
    <noServer>true</noServer>
    <port>8080</port>
    <soyc>false</soyc>
   </configuration>
  </plugin>

  <plugin>
   <groupId>com.vaadin</groupId>
   <artifactId>vaadin-maven-plugin</artifactId>
   <version>1.0.1</version>
   <executions>
    <execution>
     <configuration>
     </configuration>
     <goals>
      <goal>update-widgetset</goal>
     </goals>
    </execution>
   </executions>
  </plugin>

  <plugin>
   <groupId>org.mortbay.jetty</groupId>
   <artifactId>maven-jetty-plugin</artifactId>
   <version>6.1.24</version>
   <configuration>
    <stopPort>9966</stopPort>
    <stopKey>${jetty.stopkey}</stopKey>
    <scanIntervalSeconds>4</scanIntervalSeconds>
```

```
    <webAppConfig>
     <contextPath>${jetty.context}</contextPath>
     <baseResource  implementati-
on="org.mortbay.resource.ResourceCollection">
      <resourcesAsCSV>
       src/main/webapp,${project.build.directory}/
       ${project.build.finalName}
      </resourcesAsCSV>
     </baseResource>
    </webAppConfig>
   </configuration>
  </plugin>

  <plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>findbugs-maven-plugin</artifactId>
   <version>2.3.2-SNAPSHOT</version>
   <configuration>
    <findbugsXmlOutput>true</findbugsXmlOutput>
    <findbugsXmlWithMessages>true</findbugsXmlWithMessages>
    <xmlOutput>true</xmlOutput>
   </configuration>
  </plugin>

  <plugin>
   <groupId>org.codehaus.cargo</groupId>
   <artifactId>cargo-maven2-plugin</artifactId>
   <version>1.0.5</version>
   <configuration>
    <container>
     <containerId>tomcat6x</containerId>
     <type>remote</type>
    </container>
    <configuration>
     <type>runtime</type>
     <properties>
      <cargo.tomcat.manager.url>
       ${tomcat.url}
      </cargo.tomcat.manager.url>
     </properties>
    </configuration>
   </configuration>
  </plugin>
 </plugins>
</build>

<dependencies>
 <dependency>
  <groupId>com.vaadin</groupId>
  <artifactId>vaadin</artifactId>
  <version>6.5.0</version>
 </dependency>
 <dependency>
  <groupId>com.google.gwt</groupId>
  <artifactId>gwt-user</artifactId>
  <version>2.1.1</version>
  <scope>provided</scope>
 </dependency>
 <dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
```

```
  <version>1.2.13</version>
  <type>jar</type>
  <scope>compile</scope>
 </dependency>
</dependencies>

<distributionManagement>
 <repository>
  <id>pg-releases</id>
  <url>
   https://devserver/nexus/content/repositories/releases
  </url>
 </repository>
 <snapshotRepository>
  <id>pg-snapshots</id>
  <url>
   https://devserver/nexus/content/repositories/snapshots
  </url>
 </snapshotRepository>
</distributionManagement>

<pluginRepositories>
 <pluginRepository>
  <id>codehaus-snapshots</id>
  <url>
   https://devserver/nexus/content/repositories/codehaus-snapshots
  </url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
  <releases>
   <enabled>false</enabled>
  </releases>
 </pluginRepository>
 <pluginRepository>
  <id>vaadin-snapshots</id>
  <url>
   https://devserver/nexus/content/repositories/vaadin-snapshots
  </url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
  <releases>
   <enabled>false</enabled>
  </releases>
 </pluginRepository>
 <pluginRepository>
  <id>central</id>
  <url>
   https://devserver/nexus/content/repositories/central
  </url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
  <releases>
   <enabled>true</enabled>
  </releases>
 </pluginRepository>
</pluginRepositories>

<repositories>
```

```
 <repository>
  <id>central</id>
  <url>
   https://devserver/nexus/content/repositories/central
  </url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
  <releases>
   <updatePolicy>never</updatePolicy>
  </releases>
 </repository>
 <repository>
  <id>vaadin-snapshots</id>
  <url>
   https://devserver/nexus/content/repositories/vaadin-snapshots
  </url>
  <releases>
   <enabled>false</enabled>
  </releases>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
 </repository>
 <repository>
  <id>vaadin-addons</id>
  <url>
   https://devserver/nexus/content/repositories/vaadin-addons
  </url>
 </repository>
 <repository>
  <id>pg-thirdparty</id>
  <url>
   https://devserver/nexus/content/repositories/thirdparty
  </url>
 </repository>
 <repository>
  <id>pg-snapshots</id>
  <url>
   https://devserver/nexus/content/repositories/snapshots
  </url>
 </repository>
 <repository>
  <id>pg-releases</id>
  <url>
   https://devserver/nexus/content/repositories/releases
  </url>
 </repository>
</repositories>

</project>
```